Univerzális programozás

Így neveld a programozód!

Ed. BHAX, DEBRECEN,
2020. március 22, v0.0.7.

COLLABORATORS

	TITLE : Univerzális programozás	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Ã©s Bátfai, Margaréta	2020. június 7.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	A Brun tételes feladat kidolgozása.	nbatfai
0.0.5	2020-03-02	Az Chomsky/$a^nb^nc^n$ és Caesar/EXOR csokor feladatok kiírásának aktualizálása (a heti előadás és laborgyakorlatok támogatására).	nbatfai

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.0.6	2020-03-21	A MALMÖ projektes feladatok átvezetése, minden csokor utolsó feladata Minecraft ágensprogramozás ezzel. Mottók aktualizálása. Prog1 feladatok aktuzalizálása. Javasolt (soft skill) filmek, elméletek, könyvek, előadások be.	nbatfai
0.0.7	2020-03-22	Javítások.	nbatfai

Ajánlás

„To me, you understand something only if you can program it. (You, not someone else!) Otherwise you don't really understand it, you only think you understand it."

—Gregory Chaitin, *META MATH! The Quest for Omega*, [METAMATH]

„Csak kicsi hatást ért el a videójáték-ellenes kampány. A legtöbb iskolában kétműszakos üzemben dolgoznak a számítógépek, értő és áldozatos tanárok ellenőrzése mellett."

„Minden számítógép-pedagógus tudja a világon, hogy játékokkal kell kezdeni. A játékot követi a játékprogramok írása, majd a tananyag egyes részeinek a feldolgozása.„

—Marx György, *Magyar Tudomány, 1987 (27) 12.*, [MARX]

„I can't give complete instructions on how to learn to program here — it's a complex skill. But I can tell you that books and courses won't do it — many, maybe most of the best hackers are self-taught. You can learn language features — bits of knowledge — from books, but the mind-set that makes that knowledge into living skill can be learned only by practice and apprenticeship. What will do it is (a) reading code and (b) writing code."

—Eric S. Raymond, *How To Become A Hacker, 2001.*, http://www.catb.org/~esr/faqs/hacker-howto.html

„I'm going to work on artificial general intelligence (AGI)."

I think it is possible, enormously valuable, and that I have a non-negligible chance of making a difference there, so by a Pascal's Mugging sort of logic, I should be working on it.

For the time being at least, I am going to be going about it "Victorian Gentleman Scientist" style, pursuing my inquiries from home, and drafting my son into the work."

—John Carmack, *Facebook post, 2019.*, in his private Facebook post

Tartalomjegyzék

Ábrák jegyzéke

Előszó

Amikor programozónak terveztem állni, ellenezték a környezetemben, mondván, hogy kell szövegszerkesztő meg táblázatkezelő, de az már van... nem lesz programozói munka.

Tévedtek. Hogy egy generáció múlva kell-e még tömegesen hús-vér programozó vagy olcsóbb lesz allokálni igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Mindenesetre a programozás a teoretikus kultúra csúcsa. A GNU mozgalomban látom annak garanciáját, hogy ebben a szellemi kalandban a gyerekeim is részt vehessenek majd. Ezért programozunk.

Hogyan forgasd

A könyv célja egy stabil programozási szemlélet kialakítása az olvasóban. Módszere, hogy hetekre bontva ad egy tematikus feladatcsokrot. Minden feladathoz megadja a megoldás forráskódját és forrásokat feldolgozó videókat. Az olvasó feladata, hogy ezek tanulmányozása után maga adja meg a feladat megoldásának lényegi magyarázatát, avagy írja meg a könyvet.

Miért univerzális? Mert az olvasótól (kvázi az írótól) függ, hogy kinek szól a könyv. Alapértelmezésben gyerekeknek, mert velük készítem az iniciális változatot. Ám felhasználjuk az egyetemi programozás oktatásban is: a reguláris programozás képzésben minden hallgató otthon elvégzendő labormérési jegyzőkönyvként, vagy kollokviumi jegymegajánló dolgozatként írja meg a saját változatát belőle. Ahogy szélesedni tudna a felhasználók köre, akkor lehetne kiadása különböző korosztályú gyerekeknek, családoknak, szakköröknek, programozás kurzusoknak, felnőtt és továbbképzési műhelyeknek és sorolhatnánk...

Milyen nyelven nyomjuk?

C (mutatók), C++ (másoló és mozgató szemantika) és Java (lebutított C++) nyelvekből kell egy jó alap, ezt kell kiegészíteni pár R (vektoros szemlélet), Python (gépi tanulás bevezető), Lisp és Prolog (hogy lássuk mást is) példával.

Hogyan nyomjuk?

Rántsd le a https://gitlab.com/nbatfai/bhax git repót, vagy méginkább forkolj belőle magadnak egy sajátot a GitLabon, ha már saját könyvön dolgozol!

Ha megvannak a könyv DocBook XML forrásai, akkor az alább látható **make** parancs ellenőrzi, hogy „jól formázottak” és „érvényesek-e” ezek az XML források, majd elkészíti a dblatex programmal a könyved pdf változatát, íme:

```
batfai@entropy:~$ cd glrepos/bhax/thematic_tutorials/bhax_textbook/
batfai@entropy:~/glrepos/bhax/thematic_tutorials/bhax_textbook$ make
rm -f bhax-textbook-fdl.pdf
xmllint --xinclude bhax-textbook-fdl.xml --output output.xml
xmllint --relaxng http://docbook.org/xml/5.0/rng/docbookxi.rng output.xml  ↩
   --noout
output.xml validates
rm -f output.xml
dblatex bhax-textbook-fdl.xml -p bhax-textbook.xls
Build the book set list...
Build the listings...
XSLT stylesheets DocBook - LaTeX 2e (0.3.10)
==================================================
Stripping NS from DocBook 5/NG document.
Processing stripped document.
Image 'dblatex' not found
Build bhax-textbook-fdl.pdf
'bhax-textbook-fdl.pdf' successfully built
```

Ha minden igaz, akkor most éppen ezt a legenerált `bhax-textbook-fdl.pdf` fájlt olvasod.

A DocBook XML 5.1 új neked?
Ez esetben forgasd a https://tdg.docbook.org/tdg/5.1/ könyvet, a végén találod az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

Ne cifrázzuk: programok írása. Mik akkor a programok? Mit jelent az írásuk?

Magam is ezeken gondolkozok. Szerintem a programozás lesz a jegyünk egy másik világba..., hogy a galaxisunk közepén lévő fekete lyuk eseményhorizontjának felületével ez milyen relációban van, ha egyáltalán, hát az homályos...

1.2. Milyen doksikat olvassak el?

- Kezd ezzel: http://esr.fsf.hu/hacker-howto.html!
- Olvasgasd aztán a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- C kapcsán a [KERNIGHANRITCHIE] könyv adott részei.
- C++ kapcsán a [BMECPP] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány ISO/IEC 9899:2017 kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a The GNU C Reference Manual, mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipetek! Aki pdf-ben jobban szereti olvasni: `https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf`
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [**?**] könyv 25-49, kb. 20 oldalas gyorstalpaló részét.

1.3. Milyen filmeket, előadásokat nézzek meg, könyveket olvassak el?

A kurzus kultúrájának élvezéséhez érdekes lehet a következő elméletek megismerése, könyvek elolvasása, filmek megnézése.

Elméletek.

- Einstein: A speciális relativítás elmélete.
- Schrödinger: Mi az élet?
- Penrose-Hameroff: Orchestrated objective reduction.
- Julian Jaynes: Breakdown of the Bicameral Mind.

Könyvek.

- Carl Sagan, Kapcsolat.
- Roger Penrose, A császár új elméje.
- Asimov: Én, a robot.
- Arthur C. Clarke: A gyermekkor vége.

Előadások.

- Mariano Sigman: Your words may predict your future mental health, https://youtu.be/uTL9tm7S1Io, hihetetlen, de Julian Jaynes kétkamarás tudat elméletének legjobb bizonyítéka információtechnológiai...
- Daphne Bavelier: Your brain on video games, https://youtu.be/FktsFcooIG8, az esporttal kapcsolatos sztereotípiák eloszlatására („The video game players of tomorrow are older adults”: 0.40-1:20, „It is not true that Screen time make your eyesight worse”: 5:02).

Filmek.

- 21 - Las Vegas ostroma, https://www.imdb.com/title/tt0478087/, benne a Monty Hall probléma bemutatása.
- Rain Man, https://www.imdb.com/title/tt0095953/, az [**?**] munkát ihlette, melyeket akár az MNIST-ek helyett lehet csinálni.
- Kódjátszma, https://www.imdb.com/title/tt2084970, benne a kódtörő feladat élménye.
- Interstellar, https://www.imdb.com/title/tt0816692.
- Middle Men, https://www.imdb.com/title/tt1251757/, mitől fejlődött az internetes fizetés?
- Pixels, https://www.imdb.com/title/tt2120120/, mitől fejlődött a PC?

- Gattaca, https://www.imdb.com/title/tt0119177/.
- Snowden, https://www.imdb.com/title/tt3774114/.
- The Social Network, https://www.imdb.com/title/tt1285016/.
- The Last Starfighter, https://www.imdb.com/title/tt0087597/.
- What the #$*! Do We (K)now!?,, https://www.imdb.com/title/tt0399877/.
- I, Robot, https://www.imdb.com/title/tt0343818.

Sorozatok.

- Childhood's End, https://www.imdb.com/title/tt4171822/.
- Westworld, https://www.imdb.com/title/tt0475784/, Ford az első évad 3. részében konkrétan meg is nevezi Julian Jaynes kétkamarás tudat elméletét, mint a hosztok programozásának alapját...
- Chernobyl, https://www.imdb.com/title/tt7366338/.
- Stargate Universe, https://www.imdb.com/title/tt1286039, a Destiny célja a mikrohullámú háttér struktúrája mögötti rejtély feltárása...
- The 100, https://www.imdb.com/title/tt2661044/.
- Genius, https://www.imdb.com/title/tt5673782.

II. rész

Tematikus feladatok

Bátf41 Haxor Stream
A feladatokkal kapcsolatos élő adásokat sugároz a https://www.twitch.tv/nbatfai csatorna, melynek permanens archívuma a https://www.youtube.com/c/nbatfai csatornán található.

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

Írj olyan C végtelen ciklusokat, amelyek 0 illetve 100 százalékban dolgoztatnak egy magot és egy olyat, amely 100 százalékban minden magot!

Megoldás videó: https://youtu.be/lvmi6tyz-nI

Megoldás forrása: `turing/spin.c`, `turing/idle.c`, `turing/spin_all_cores.c`.

Számos módon hozhatunk és hozunk létre végtelen ciklusokat. Vannak esetek, amikor ez a célunk, például egy szerverfolyamat fusson folyamatosan és van amikor egy bug, mert ott lesz végtelen ciklus, ahol nem akartunk. Saját példáinkban ilyen amikor a PageRank algoritmus rázza az 1 liter vizet az internetben, de az iteráció csak nem akar konvergálni...

Az alábbi megoldások a végtelen ciklusok mellett tartalmaznak egy pár Linuxos rendszerhívást is, ezek azt a célt szolgálják, hogy az ütemező ténylegesen csak egy processzor magot adjon a folyamatunknak, vagy éppen az összeset.

Egy mag 100 százalékban:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sched.h> //Linux-specifikus fejléc, ütemezéssel kapcsolatos ↩
   hívásokat tartalmaz

int main(int argc, char **argv)
{
    cpu_set_t cpus;
    CPU_ZERO(&cpus);
    CPU_SET(0, &cpus);
    sched_setaffinity(0, sizeof(cpus), &cpus); //Csak a 0-ás számú magon ↩
       akarunk futni

    while(1);
```

```
    return 0;
}
```

Egy mag 0 százalékban

Ez ugyan nem egy ciklus, de gyakorlatban a sleep()-pel megoldott várakozások helyett szinte mindig van jobb opció (persze, ha adott ideig akarunk várakozni, akkor rendben van). Várhatunk szinkronizálási objektumokra, szignálokra, IO eseményekre, stb... Ez a példa a SIGINT szignált várja:

```
#define _GNU_SOURCE
#include <signal.h>

int main(int argc, char **argv)
{
    sigset_t sigset;
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGINT);

    int sig;
    sigwait(&sigset, &sig); //Várjuk a SIGINT jelre (Ctrl+C)

    return 0;
}
```

Minden mag 100 százalékban (az OMP-s megoldásnál kicsit explicitebb módon, POSIX szálakkal; persze az eredmény lényegében ugyanaz):

```
#define _GNU_SOURCE
#include <unistd.h>
#include <pthread.h>
#include <sys/sysinfo.h>

void spin(int proc)
{
    cpu_set_t cpus;
    CPU_ZERO(&cpus);
    CPU_SET(proc, &cpus);
    pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpus);

    while(1);
}

int main(int argc, char **argv)
{
    int nprocs = get_nprocs_conf(); //Lekérdezzük a processzorok számát ( ↩
       ehhez kell a sys/sysinfo.h)

    pthread_t threads[nprocs];
```

```
    for(int i = 0; i < nprocs; i++)
        pthread_create(&threads[i], NULL, spin, &i);

    pthread_join(threads[0], NULL);

    return 0;
}
```

Ha a fenti programokra ránézünk a **top** parancs kimenetében, láthatjuk, hogy tényleg a várt módon használják a CPU-t.

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
11238 gabriel+  20   0    2144    692    628 R 100.0   0.0   2:06.51 spin
11351 gabriel+  20   0    2144    752    688 S   0.0   0.0   0:00.00 idle
11311 gabriel+  20   0   35204    728    648 S 396.2   0.0   0:47.99 spin_all_+
```

Werkfilm

- https://youtu.be/lvmi6tyz-nI

2.2. Lefagyott, nem fagyott, akkor most mi van?

Mutasd meg, hogy nem lehet olyan programot írni, amely bármely más programról eldönti, hogy le fog-e fagyni vagy sem!

Megoldás videó:

Megoldás forrása: tegyük fel, hogy akkora haxorok vagyunk, hogy meg tudjuk írni a `Lefagy` függvényt, amely tetszőleges programról el tudja dönteni, hogy van-e benne vlgtelen ciklus:

```
Program T100
{

  boolean Lefagy(Program P)
  {
     if(P-ben van végtelen ciklus)
      return true;
     else
      return false;
  }

  main(Input Q)
  {
    Lefagy(Q)
  }
}
```

A program futtatása, például akár az előző `v.c` ilyen pszeudókódjára:

```
T100(t.c.pseudo)
true
```

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

A T100-as programot felhasználva készítsük most el az alábbi T1000-set, amelyben a Lefagy-ra épőlő Lefagy2 már nem tartalmaz feltételezett, csak csak konkrét kódot:

```
Program T1000
{

  boolean Lefagy(Program P)
  {
     if(P-ben van végtelen ciklus)
      return true;
     else
      return false;
  }

  boolean Lefagy2(Program P)
  {
     if(Lefagy(P))
      return true;
     else
      for(;;);
  }

  main(Input Q)
  {
    Lefagy2(Q)
  }

}
```

Mit for kiírni erre a `T1000(T1000)` futtatásra?

- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true
- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

akkor most hogy fog működni? Sehogy, mert ilyen `Lefagy` függvényt, azaz a T100 program nem is létezik.

Tanulságok, tapasztalatok, magyarázat...

2.3. Változók értékének felcserélése

Írj olyan C programot, amely felcseréli két változó értékét, bármiféle logikai utasítás vagy kifejezés használata nélkül!

Megoldás videó: https://bhaxor.blog.hu/2018/08/28/10_begin_goto_20_avagy_elindulunk

A megszokott megoldás két változó értékének felcserélésére egy segédváltozó használata:

```
int a, b;
/* ... */

int c = a;
a = b;
b = c;
```

Bizonyos típusú változókat kivonással is felcserélhetünk:

```
a -= b;
b += a;
a = b - a;
```

Egy még érdekesebb (és a legtöbb típusra alkalmazható) megoldás a bitenkénti kizáró vagy:

```
a ^= b;
b ^= a;
a ^= b;
```

Ezek mellett van lehetőségünk stringek használatára (noha ez ilyen formában a valóságban teljesen haszontalan):

```
char buffer[32];

snprintf(buffer, 32, "%d %d", b, a);
sscanf(buffer, "%d %d", &a, &b);
```

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés nasználata nélkül írj egy olyan programot, ami egy labdát pattogtat a karakteres konzolon! (Hogy mit értek pattogtatás alatt, alább láthatod a videókon.)

Megoldás videó: https://bhaxor.blog.hu/2018/08/28/labdapattogas

Tanulságok, tapasztalatok, magyarázat...

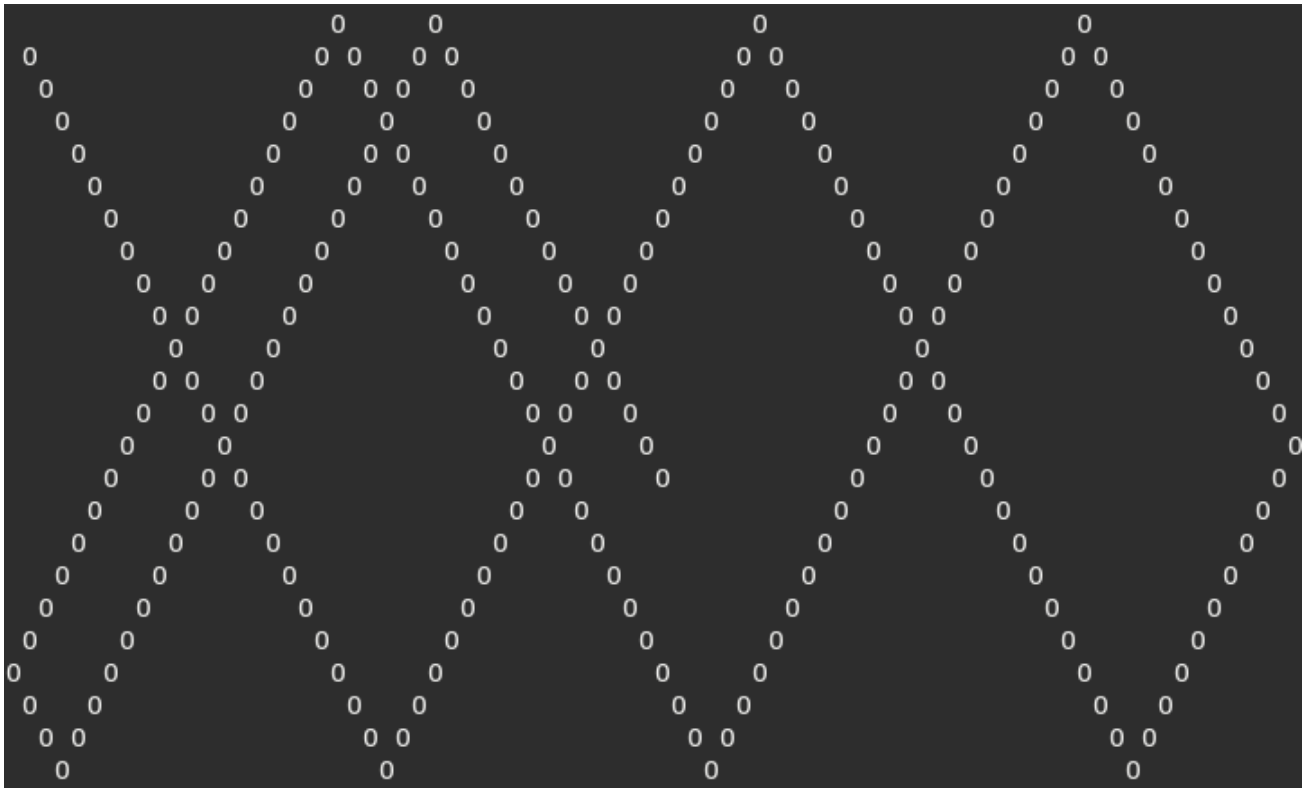
A feladat megoldása if-ekkel

```
#include <unistd.h>
#include <ncurses.h>
#include <math.h>

int main(int argc, char **argv)
{
    initscr();
    noecho();
    curs_set(0);

    int rows, cols, pos_x, pos_y, vel_x, vel_y;

    pos_x = pos_y = 0;
    vel_x = vel_y = 1;

    while(TRUE)
    {
        pos_x += vel_x;
        pos_y += vel_y;

        getmaxyx(stdscr, rows, cols); //Ha netán átméretezik a terminált,  ↩
           ne okozzon gondot
        if(pos_x > cols - 1) pos_x = cols - 1;
        if(pos_y > rows - 1) pos_y = rows - 1;

        if(pos_x < 1) vel_x *= -1;
        if(pos_y < 1) vel_y *= -1;
        if(pos_x >= cols - 1) vel_x *= -1;
        if(pos_y >= rows - 1) vel_y *= -1;


        clear();

        mvaddch(pos_y, pos_x, 'O');

        refresh();
        usleep(100000);
    }
```

```
    endwin();

    return 0;
}
```

Már ebben az egyszerű példában megjelenik a játékoknál (és nem is csak játékoknál) igen gyakori "main loop", ugyanis a programunk lényege egy cikluson belül van. Nagyon sok játéknak ugyanis végsősoron az a célja, hogy másodpercenként (optimális esetben) ~60 képkockát megrajzoljon, és két rajzolás közben ellenőrizze végigmenjen a játék logikáján. A mi esetünkben is valami ilyesmi történik, a while cikluson belül először léptetjük a labda pozícióját és frissítjük a sebességvektorát, majd rajzoljuk a képernyőre.

Itt egy képernyőkép a programról, és, hogy szemléletesebb legyen, kivettem belőle a képernyő törlését (clear függvény):



Ugyanez if-ek nélkül

Ha nem használhatunk elágazásokat, újra kell gondolnunk a program azon részét, amely a labda pozíciójának határaival és a labda sebességével foglalkoznak:

```
getmaxyx(stdscr, rows, cols);
if(pos_x > cols - 1) pos_x = cols - 1;
if(pos_y > rows - 1) pos_y = rows - 1;

if(pos_x < 1) vel_x *= -1;
if(pos_y < 1) vel_y *= -1;
if(pos_x >= cols - 1) vel_x *= -1;
if(pos_y >= rows - 1) vel_y *= -1;
```

Az átméretezéses részt cseréljük le egy modulo alapú megoldásra:

```
getmaxyx(stdscr, rows, cols);
pos_x %= cols;
pos_y %= rows;
```

A sebesség változtatását (a tényleges "pattogást") pedig oldjuk meg úgy, hogy a sebességvektor komponenseit megszorozzuk valamivel minden iterációban. Ennek a valaminek 1-nek kéne lennie alapesetben, és -1-nek, ha a terminál szélén vagyunk (tengelyenként külön véve). Vezessünk be ehhez egy függvényt:

```
vel_x *= is_within_bounds(pos_x, cols - 1);
vel_y *= is_within_bounds(pos_y, rows - 1);
```

Már csak ezt a függvényt kell implementálni. Sok megoldást ki lehet találni a problémára, ez a változat az integerosztás sajátosságait használja fel:

```
//A visszatérési érték -1, ha n<=0 vagy n>=max, egyébként pedig 1
int is_within_bounds(int n, int max)
{
    int upper = (n / max) * -2 + 1;
    int lower = ((max - n) / max) * -2 + 1;

    return upper * lower;
}
```

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Írj egy programot, ami megnézi, hogy hány bites a szó a gépeden, azaz mekkora az int mérete. Használd ugyanazt a while ciklus fejet, amit Linus Torvalds a BogoMIPS rutinjában!

Megoldás videó: https://youtu.be/9KnMqrkj_kU, https://youtu.be/KRZlt1ZJ3qk, .

A szóhossz, mint fogalom, alapvetően a jelen hardver "adatbuszának" a méretére utal (data bus - rémes dolog magyarul informatikáról beszélni). Ezt szoftveren keresztül nem triviális lekérdezni, bár rendszerinformációkon át el lehet hozzá jutni, de közvetlenül nem igazán van mód rá.

Átfogalmazva a problémát, a gépi szó hossza megfelel annak az adatmennyiségnek, mellyel a processzor alapvetően egyszerre dolgozni tud (kerüljük ki a Single Instruction Multiple Data kérdéskörét), ami általában célszerűen egybeesik a főbb regiszterek méretével egy adott architektúrán. Ezt még mindg nem deríthető ki egyszerűen...

A sizeof(int) saját olvasatom szerint nem ad pontos képet a szóhosszról, többek között az én gépemen sem ad helyes választ. Egy jobb megoldás a pointerek méretének vizsgálata, mivel sok esetben a címek hossza gyakran megegyezik a gépi szó méretével (bár még ez a megoldás sem teljesen pontos).

```
printf("sizeof(void*) = %zu .. * 8 = %u bits\n\n", sizeof(void*), (unsigned ↩
   )sizeof(void*) * 8);
```

Rátérve a BogoMIPSre, a program alapja az alábbi függvény, amely egy üres ciklusnak megadott számú iterációját hajtja végre. A BOGOMips alapvetően kalibrálási célt szolgál (elsősorban az udelay/mdelay busy wait függvényekhez, régebben nagyobb szerepük volt), különböző processzorok összehasonlítására nem jó hasonlítási alap (nem véletlen a "bogo" előtag). Így esetünkben lényegtelen, hogy ez a ciklus pontosan, instrukcióról instrukcióra hogyan épül fel, és hogy adott architektúrán hogyan fut le.

```
void spin(unsigned long long iter)
{
    for(; iter > 0; iter--);
}
```

A `main()`-ben ezután, nagyságrendekben gondolkodva (2 hatványai), megkeressük a legkisebb `loops_per_se` értéket, amelynél a `spin()` már tovább fut, mint 1 másodperc. Ha ez megvan, becslésként a kapott értéket elosztjuk a ténylegesen eltelt idővel.

```
int main(int argc, char **argv)
{
    printf("*** Ténylegesen pontos BOGOMips értékekhez keresd fel a /proc/ ↩
       cpuinfo-t! ***\n");

    unsigned long long loops_per_sec = 1;
    unsigned long long mcsecs;

    while(loops_per_sec <<= 1)
    {
        mcsecs = clock();
        spin(loops_per_sec);
        mcsecs = clock() - mcsecs;

        if (mcsecs >= 1000000)
        {
            loops_per_sec = loops_per_sec / mcsecs * 1000000;

            printf("BOGOMips: %llu\n", loops_per_sec / 100000);

            return 0;
        }
    }

    return -1;
}
```

2.6. Helló, Google!

Írj olyan C programot, amely egy 4 honlapból álló hálózatra kiszámolja a négy lap Page-Rank értékét!

Megoldás videó:

A PageRank algoritmus volt a Google óriási sikerének egyik nagy tényezője. Egy igen elegáns, és matematikailag robusztus megoldást kínál egy sokoldalú problémára az internetes keresés kérdéskörében:

Mely keresési eredmények jelenjenek meg az első találatok között?

A PageRank esetében ezt a problémát úgy közelítjük meg, hogy megvizsgáljuk az oldalakon található linkeket, pontosabban azt, hogy ezen linkeket véletlenszerűen követve melyik oldalon mekkora eséllyel lyukadunk ki. Furcsának tűnhet elsőre, de az így kapott valószínűségeknek máig is fontos szerepük van a keresőmotorok működésében (a neurális hálók egyre nagyobb térnyerése mellett).

A módszer sikeréhez nagyban hozzájárul az, hogy mindez leképezhető mátrix sajátvektorainak kiszámítására, melyeket egészen egyszerű iteratív módszerekkel megközelíteni.

Az alábbi források részletesebben összefoglalják az elméleti hátteret:

- http://home.ie.cuhk.edu.hk/~wkshum/papers/pagerank.pdf
- https://www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm (egy kissé eltérő megoldás)

A megoldásomat megpróbáltam úgy felépíteni, hogy szépen látszódjon az elméleti anyagban leírt `x = Ax` iterációja:

```
int main(void)
{
    /*
        M[i][j] értéke aszerint alakul, hogy
        a j sorszámú oldalon van-e az
        i sorszámú oldalra mutató link.
        M oszlopainak az összege 1 (oszlop-sztochasztikus mtrix)
    */
    double M[4][4] = {
        {0.0, 0.0, 0.33, 0.0},
        {1.0, 0.0, 0.33, 1.0},
        {0.0, 0.5, 0.00, 0.0},
        {0.0, 0.5, 0.33, 0.0}
    };

    //"simítunk" a mátrixon (damping); ez egy egész gyakori módszer
    //a "lógó" oldalak által okozott végeten ciklusok elkerülésére ( ↩
       részletek az első forrásban)
    damp_matrix(4, M, 0.85);

    double PR[4] = {1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0};
    double PR_prev[4] = {0.0, 0.0, 0.0, 0.0};
```

```
    while(1)
    {
        if (delta(4, PR, PR_prev) < 0.000001)
            break;

        //PageRank iteration:
        Amulx(4, M, PR, PR_prev);
        swap(PR, PR_prev, sizeof(PR));
    }

    return 0;
}
```

Ahol az `Amulx(n, A, x, result)` kiszámítja az A mátrix és az x vektor szorzatát és a `delta(n, vec, prev_vec)` visszaadja két vektor különbségének a hosszát. A felhasznált függvények az implementációi:

```
void damp_matrix(int size, double mat[size][size], double d)
{
    for(int i = 0; i < size; i++)
        for(int j = 0; j < size; j++)
            mat[i][j] = d * mat[i][j] + (1.0 - d) / size;
}

void Amulx(int n, double A[n][n], double x[n], double result[n])
{
    for(int i = 0; i < n; i++)
    {
        result[i] = 0;
        for(int j = 0; j < n; j++)
            result[i] += A[i][j] * x[j];
    }
}

double delta(int n, double vec[n], double prev_vec[n])
{
    double sum = 0.0;
    for (int i = 0; i < n; i++)
        sum += (vec[i] - prev_vec[i]) * (vec[i] - prev_vec[i]);
    sum /= n;

    return sqrt(sum);
}


void swap(void *a, void *b, size_t len)
{
```

```
    char tmp;
    char *ach = (char*)a;
    char *bch = (char*)b;

    for(size_t i = 0; i < len; i++)
    {
        tmp = ach[i];
        ach[i] = bch[i];
        bch[i] = tmp;
    }
}

void print_array(int n, double array[n])
{
    for (int i = 0; i < n; i++)
        printf("%f ", array[i]);
    printf("\n");
}
```

2.7. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Megoldás videó: https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/MontyHall_R

Tanulságok, tapasztalatok, magyarázat...

2.8. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

Megoldás videó: https://youtu.be/xbYhp9G6VqQ

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/Primek_R

A természetes számok építőelemei a prímszámok. Abban az értelemben, hogy minden természetes szám előállítható prímszámok szorzataként. Például 12=2*2*3, vagy például 33=3*11.

Prímszám az a természetes szám, amely csak önmagával és eggyel osztható. Eukleidész görög matematikus már Krisztus előtt tudta, hogy végtelen sok prímszám van, de ma sem tudja senki, hogy végtelen sok ikerprím van-e. Két prím ikerprím, ha különbségük 2.

Két egymást követő páratlan prím között a legkisebb távolság a 2, a legnagyobb távolság viszont bármilyen nagy lehet! Ez utóbbit könnyű bebizonyítani. Legyen n egy tetszőlegesen nagy szám. Akkor szorozzuk össze n+1-ig a számokat, azaz számoljuk ki az 1*2*3*... *(n-1)*n*(n+1) szorzatot, aminek a neve (n+1) faktoriális, jele (n+1)!.

Majd vizsgáljuk meg az a sorozatot:

(n+1)!+2, (n+1)!+3,... , (n+1)!+n, (n+1)!+ (n+1) ez n db egymást követő azám, ezekre (a jól ismert bizonyítás szerint) rendre igaz, hogy

- (n+1)!+2=1*2*3*... *(n-1)*n*(n+1)+2, azaz 2*valamennyi+2, 2 többszöröse, így ami osztható kettővel
- (n+1)!+3=1*2*3*... *(n-1)*n*(n+1)+3, azaz 3*valamennyi+3, ami osztható hárommal
- ...
- (n+1)!+(n-1)=1*2*3*... *(n-1)*n*(n+1)+(n-1), azaz (n-1)*valamennyi+(n-1), ami osztható (n-1)-el
- (n+1)!+n=1*2*3*... *(n-1)*n*(n+1)+n, azaz n*valamennyi+n-, ami osztható n-el
- (n+1)!+(n+1)=1*2*3*... *(n-1)*n*(n+1)+(n-1), azaz (n+1)*valamennyi+(n+1), ami osztható (n+1)-el

tehát ebben a sorozatban egy prim nincs, akkor a (n+1)!+2-nél kisebb első prim és a (n+1)!+ (n+1)-nél nagyobb első prim között a távolság legalább n.

Az ikerprímszám sejtés azzal foglalkozik, amikor a prímek közötti távolság 2. Azt mondja, hogy az egymástól 2 távolságra lévő prímek végtelen sokan vannak.

A Brun tétel azt mondja, hogy az ikerprímszámok reciprokaiból képzett sor összege, azaz a (1/3+1/5)+ (1/5+1/7)+ (1/11+1/13)+... véges vagy végtelen sor konvergens, ami azt jelenti, hogy ezek a törtek összeadva egy határt adnak ki pontosan vagy azt át nem lépve növekednek, ami határ számot B_2 Brun konstansnak neveznek. Tehát ez nem dönti el a több ezer éve nyitott kérdést, hogy az ikerprímszámok halmaza végtelen-e? Hiszen ha véges sok van és ezek reciprokait összeadjuk, akkor ugyanúgy nem lépjük át a B_2 Brun konstans értékét, mintha végtelen sok lenne, de ezek már csak olyan csökkenő mértékben járulnának hozzá a végtelen sor összegéhez, hogy így sem lépnék át a Brun konstans értékét.

Ebben a példában egy olyan programot készítettünk, amely közelíteni próbálja a Brun konstans értékét. A repó bhax/attention_raising/Primek_R/stp.r mevű állománya kiszámolja az ikerprímeket, összegzi a reciprokaikat és vizualizálja a kapott részeredményt.

```
#    Copyright (C) 2019  Dr. Norbert Bátfai, nbatfai@gmail.com
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <http://www.gnu.org/licenses/>

library(matlab)
```

```
stp <- function(x){

    primes = primes(x)
    diff = primes[2:length(primes)]-primes[1:length(primes)-1]
    idx = which(diff==2)
    t1primes = primes[idx]
    t2primes = primes[idx]+2
    rt1plust2 = 1/t1primes+1/t2primes
    return(sum(rt1plust2))
}

x=seq(13, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```

Soronként értelemezzük ezt a programot:

```
 primes = primes(13)
```

Kiszámolja a megadott számig a prímeket.

```
> primes=primes(13)
> primes
[1]  2  3  5  7 11 13
```

```
 diff = primes[2:length(primes)]-primes[1:length(primes)-1]
```

```
> diff = primes[2:length(primes)]-primes[1:length(primes)-1]
> diff
[1] 1 2 2 4 2
```

Az egymást követő prímek különbségét képzi, tehát 3-2, 5-3, 7-5, 11-7, 13-11.

```
idx = which(diff==2)
```

```
> idx = which(diff==2)
> idx
[1] 2 3 5
```

Megnézi a `diff`-ben, hogy melyiknél lett kettő az eredmény, mert azok az ikerprím párok, ahol ez igaz. Ez a `diff`-ben lévő 3-2, 5-3, 7-5, 11-7, 13-11 különbségek közül ez a 2., 3. és 5. indexűre teljesül.

```
t1primes = primes[idx]
```

Kivette a primes-ból a párok első tagját.

```
t2primes = primes[idx]+2
```

A párok második tagját az első tagok kettő hozzáadásával képezzük.

```
rt1plust2 = 1/t1primes+1/t2primes
```

Az 1/t1primes a t1primes 3,5,11 értékéből az alábbi reciprokokat képzi:

```
> 1/t1primes
[1] 0.33333333 0.20000000 0.09090909
```

Az 1/t2primes a t2primes 5,7,13 értékéből az alábbi reciprokokat képzi:

```
> 1/t2primes
[1] 0.20000000 0.14285714 0.07692308
```

Az 1/t1primes + 1/t2primes pedig ezeket a törteket rendre összeadja.

```
> 1/t1primes+1/t2primes
[1] 0.5333333 0.3428571 0.1678322
```

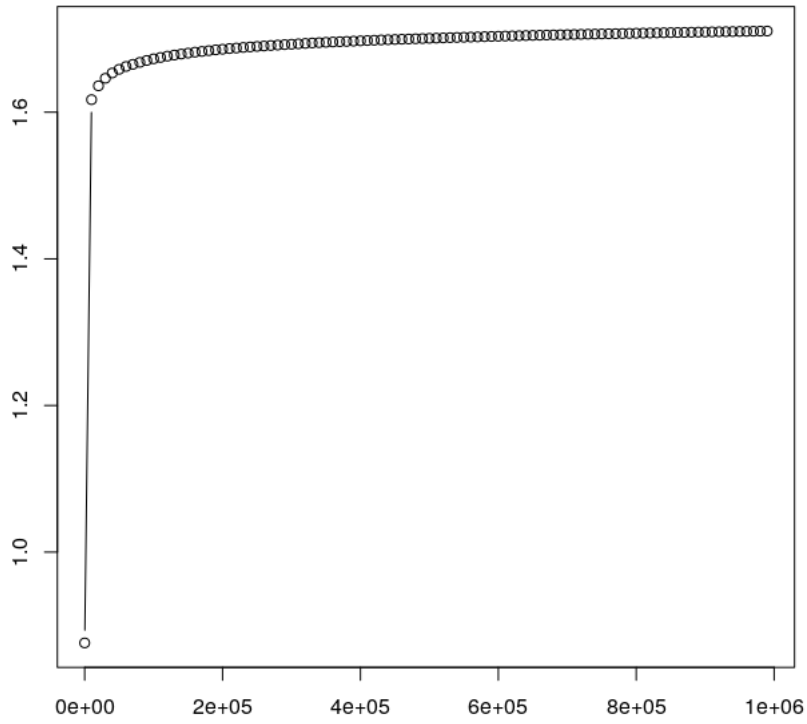
Nincs más dolgunk, mint ezeket a törteket összeadni a `sum` függvénnyel.

```
sum(rt1plust2)
```

```
>   sum(rt1plust2)
[1] 1.044023
```

A következő ábra azt mutatja, hogy a szumma értéke, hogyan nő, egy határértékhez tart, a B_2 Brun konstanshoz. Ezt ezzel a csipettel rajzoltuk ki, ahol először a fenti számítást 13-ig végezzük, majd 10013, majd 20013-ig, egészen 990013-ig, azaz közel 1 millióig. Vegyük észre, hogy az ábra első köre, a 13 értékhez tartozó 1.044023.

```
x=seq(13, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```



2.1. ábra. A B_2 konstans közelítése

Werkfilm

- https://youtu.be/VkMFrgBhN1g
- https://youtu.be/aF4YK6mBwf4

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás forrása: az első előadás 27 fólia.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

3.2. Az $a^nb^nc^n$ nyelv nem környezetfüggetlen

Mutass be legalább két környezetfüggő generatív grammatikát, amely ezt a nyelvet generálja!

Megoldás forrása: az első előadás 30-32 fólia.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

3.3. Hivatkozási nyelv

A [KERNIGHANRITCHIE] könyv C referencia-kézikönyv/Utasítások melléklete alapján definiáld BNF-ben a C utasítás fogalmát! Majd mutass be olyan kódcsipeteket, amelyek adott szabvánnyal nem fordulnak (például C89), mással (például C99) igen.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

Ugyan ez nem BNF alakban van, de érdekességként itt van a KR könyv első kiadásában leírt C utasítás, ahogy eredetileg meg van írva:

```
statement:
    compound-statement
    expression ;
    if ( expression ) statement
    if ( expression ) statement else statement       //Megjegyzés: nincs  ↩
       szükség külön else if-re!
    while ( expression ) statement
    do statement while ( expression ) ;
    for ( expression<opt> ; expression<opt> ; expression<opt> ) statement  ↩
                  //
    switch ( expression ) statement
    case constant-expression : statement
    default : statement
    break ;                                                  //nincs  ↩
       többszörös break
    continue ;
    return ;
    return expression ;
    goto identifier ;
    identifier : statement
    ;

compound-statement:
    { declaration-list<opt> statement-list<opt> }                // ↩
       deklarációk csak a blokk elején

declaration_list:
    declaration
    declaration declaration_list

statement_list:
    statement
    statement statement_list
```

A 10 évvel később ANSI C szabvány utasítások terén nem hozott változásokat (máshol persze igen), mindössze a szintaxis leírása változott meg, így már kevésbé ömlesztett:

```
statement:
    labeled_statement
    compound_statement
    expression_statement
    selection_statement
    iteration_statement
    jump_statement

labeled_statement:
```

```
    identifier : statement
    case constant_expression : statement
    default : statement

compound_statement:
    { declaration_list<opt> statement_list<opt> }

declaration_list:
    declaration
    declaration_list declaration

statement_list:
    statement
    statement_list statement

expression_statement:
    ;
    expression ;

selection_statement:
    if ( expression ) statement
    if ( expression ) statement else statement
    switch ( expression ) statement

iteration_statement:
    while ( expression ) statement
    do statement WHILE ( expression ) ;
    for ( expression_statement expression_statement ) statement
    for ( expression_statement expression_statement expression ) statement

jump_statement:
    goto identifier ;
    continue ;
    break ;
    return ;
    return expression ;
```

Ami ebből elsőre talán a legfeltűnőbb, az az összetett utasítások definíciója. Eredetileg ugyanis egy utasításblokkon belül deklarációk csak a blokk elején szerepelettek (illetve függvények argumentumlistájában). Ez C99-től változott meg, onnantól szabadon lehet keverni a deklarációkat és utasításokat egy összetett utasításon belül. Ez főként a C++ hatására változott meg.

A másik tipikus példa a for ciklusok definíciója: for ciklusok zárójelei között az első tag csak C99-től lehet deklaráció is, korábban nem. Itt a C99-es definíció:

```
iteration_statement:
    while ( expression ) statement
```

```
do statement while ( expression ) ;
for ( expression_statement expression_statement ) statement
for ( expression_statement expression_statement expression ) statement
for ( declaration expression_statement ) statement
for ( declaration expression_statement expression ) statement
```

További példák standardek közti eltérésekre:

- Kommentek: C89-ben csak `/* */`, utána már sorvégi kommenteknél `//` is
- Argumentumlisták:

```
//KR C-ben még így néztek ki:

void func(a, b)
int a;
char *b;
{
    /*...*/
}

//C89-től viszont már így:

void func(int a, char *b)
{
    /*...*/
}
```

(Érdekesség: ha csak em kényszerítjük egy adott standard használatára, a GCC máig elfogadja az első változatot is kompatibilitás okán)

- Deklarációk C99 előtt csak blokkok elején, illetve függvények argumentumlistáiban lehettek
- Makrók változó számú argumentummal (C99):

```
#define macro_print(...) printf(__VA_ARGS__);
```

- `inline` függvények (C99-től)
- VLA-k (Variable Length Array): nem konstans a deklarációban megadott méretük (C99 - se előtte, se utána nem standard)

```
int n = get_some_value();
int arr[n];
```

- Változó méretű tömbök struktúrák végén (flexible array members):

```
struct buffer
{
    unsigned int len;
    unsigned char buf[];
};

struct buffer *foo = malloc(sizeof(struct buffer) + foo_len);
```

- `restrict` kulcsszó

3.4. Saját lexikális elemző

Írj olyan programot, ami számolja a bemenetén megjelenő valós számokat! Nem elfogadható olyan megoldás, amely maga olvassa betűnként a bemenetet, a feladat lényege, hogy lexert használjunk, azaz óriások vállán álljunk és ne kispályázzunk!

Megoldás videó: https://youtu.be/9KnMqrkj_kU (15:01-től).

Megoldás forrása: `bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/realnumber.l`

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

A Lex(/Flex) segítségével igen könnyen generálhatunk C lexikális elemzőket. Bár ehhez a feladathoz talán túlzás nagyágyúkat használni, bevezetésnek hasznos lehet. A lexerünk forrása három `%%`-al elválasztott részből áll, és az alábbiak szerint épül fel:

```
%{
    /* C kód (tipikusan include-ok és szükséges deklarációk) */
%}

/* Definíciók, lexer opciók */

%%
```

```
/* Szabályok */

%%

/* Szokásos C kód; a main() és minden egyéb ide kerül */
```

Az első részben include-olunk, deklarálunk két globális változót, és reguláris kifejezésekkel definiálunk két mintát (mindössze a későbbi olvashatóság érdekében):

(A program hexadecimális értékeket is keres, miért is ne.)

```
%{
#include <stdio.h>
#include <stdlib.h>
int realnums = 0, hexnums = 0;
%}

real_literal    0[xX]([0-9A-F]+|[0-9a-f]+)
hex_literal     [+-]?[0-9]*\.?[0-9]+
```

A második részben szabályokat határozunk meg, ezt regexekkel, illetve fentebb definiált mintákkal tehetjük meg. A sor a minta leírásával kezdődik, majd utána egy C kód blokkal, ami megadja, hogy az adott minta megtalálásakor mit akarunk csinálni (A `yytext` a jelenleg megtalált minta szövege). (Megjegyzésként, leggyakrabban lexereket "tokenizálásra" használunk, ilyenkor a kód általában `{ return <szimbolikus konstans>; }` alakú.)

```
{real_literal}      { hexnums++; printf("hex  number found: %s %ld\n",  ↩
   yytext, strtol(yytext, NULL, 0)); }
{hex_literal}       { realnums++; printf("real number found: %s %f\n",  ↩
   yytext, atof(yytext)); }
.|\n ;
```

A harmadik részben már csak egy main()-re van szükségünk. Itt egyszerűen meghívjuk a lexerünket (`yylex()`), majd amikor az visszatér (a bemenet végén), kiírjuk, mennyi valós számot/hex számot találtunk.

```
int main(int argc, char **argv)
{
    yylex();
```

```
    printf("number of reals in input: %d\nnumber of hex values in input: %d ↩
       \n", realnums, hexnums);

    return 0;
}
```

3.5. Leetspeak

Lexelj össze egy l33t ciphert!

Megoldás videó: https://youtu.be/06C_PqDpD_k

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/l337d1c7.l

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

```
%{
#include<stdio.h>
#include<ctype.h>

char from[20] = {'w',      'e', 't', 'z', 'i', 'o', 'p',  'a', 's', 'd',  ' ↩
   g', 'h',   'k',  'l', 'x', 'c',  'v',   'b',  'n',    'm'};
char *to[20] =  {"\\/\\/", "3", "7", "2", "1", "0", "|\textdegree{}", "4",  ↩
   "5", "|)", "9", "|-|", "|<", "1", "><", "(", "\\/", "|3", "|\\|", "|\\/| ↩
   "};
%}

%%

. {
    int found = 0;
    for(int i = 0; i < 20; i++)
    {
        if(tolower(*yytext) == from[i])
        {
            printf("%s", to[i]);
            found = 1;
            break;
        }
    }
    if(!found) printf("%c", *yytext);
}

%%
```

```
int main(int argc, char **argv)
{
    yylex();

    return 0;
}
```

3.6. A források olvasása

Hogyan olvasod, hogyan értelmezed természetes nyelven az alábbi kódcsipeteket? Például

```
if(signal(SIGINT, jelkezelo)==SIG_IGN)
    signal(SIGINT, SIG_IGN);
```

Ha a SIGINT jel kezelése figyelmen kívül volt hagyva, akkor ezen túl is legyen figyelmen kívül hagyva, ha nem volt figyelmen kívül hagyva, akkor a jelkezelo függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)

Bugok
Vigyázz, sok csipet kerülendő, mert bugokat visz a kódba! Melyek ezek és miért? Ha nem megy ránézésre, elkapja valamelyiket esetleg a splint vagy a frama?

Megoldás videó: BHAX 357. adás.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

1.
```
if(signal(SIGINT, jelkezelo)==SIG_IGN)
    signal(SIGINT, SIG_IGN);
```

   Ha a SIGINT jelet ezelőtt figyelmen kívül hagytuk, ezután is ignorájuk, ha pedig volt jelkezelő megadva hozzá, azt lecseréljük a jelkezelo függvényre

2.
```
if(signal(SIGINT, SIG_IGN)!=SIG_IGN)
    signal(SIGINT, jelkezelo);
```

   Ugyanaz, mint az előző. A signal(...) függvény a megadott szignál előző jelkezelő függvényével tér vissza (függvénypointer).

3.

```
for(i=0; i<5; ++i)
```

Szokásos for ciklus, ötször fut le, i első értéke 0, utolsó értéke 4.

4.

```
for(i=0; i<5; i++)
```

Ugyanaz, mint az előző. A preincrement/postincrement között ezesetben nincs különbség, mivel az i++ és ++i kifejezéseknek egyedül az értéke különbözik. (Mely értékkel ezesetben nem kezdünk semmit)

Szemléltetésképp, a fenti for ciklus a következő kóddal egyenértékű:

```
i=0;
loop_begin:
if(!(i<5)) goto loop_end;

{/*ciklusmag*/}

i++;
goto loop_begin;
loop_end:
;
```

5.

```
for(i=0; i<5; tomb[i] = i++)
```

Ez a csipet meghatározatlan viselkedéshez vezet (undefined behavior), mivel a C standardek (itt teljesen mindegy, melyikről beszélünk) nem határozzák meg a részkifejezések értelmezésének helyes sorrendjét (néhány kivételtől eltekintve, pl. &&, ||).

Így, ha szemléletesen megpróbáljuk kibontani a `tomb[i] = i++` értékadó kifejezést, több helyes változatot is kaphatunk:

```
tomb[i] = i;
i += 1;
```

```
int rhs = i++;
tomb[i] = rhs;
```

Ezek a változatok viszont eltérő eredményhez vezetnek.

6.

```
for(i=0; i<n && (*d++ = *s++); ++i)
```

Ez tipikus esete lehet a "== helyett használt =" elírásnak; így is lefordítható, de nincs sok haszna.

Ezen felül, az előző példához hasonlóan meghatározatlan a ++-ok kiértékelésének sorrendje, amelyek ráadásul itt mutatókat inkrementálnak (az operátorok erősorrendje alapján). Bármit is akarna jelenteni ez a csipet, ilyen formában rengeteg veszélyes hibához vezethet.

7.

```
printf("%d %d", f(a, ++a), f(++a, a));
```

Két függvényértékként kapott integert írunk ki... A gond csak azzal van, hogy, ismét a ++ miatt, meghatározatlan, hogy ezek a függvények milyen értékeket kapnak paraméterként.

Megjegyzésként, több csipetben is láttuk, hogy bizonyos operátorok "mellékhatásai" (a kifejezés változat a program állapotán) sok kellemetlenséget okozhatnak. Minden esetben érdemes odafigyelni, hogy a program mely pontjai hagynak változásokat maguk után, főleg, hogyha ezt rejtve teszik meg (pl. ha egy függvény változtat egy mutatón keresztül átadott változó értékén). Ilyen esetek egyértelműsítéséhez nyújt segítséget a `const` kulcsszó. (Főleg C++-ban, lásd const correctness)

8.

```
printf("%d %d", f(a), a);
```

9.

```
printf("%d %d", f(&a), a);
```

Mindaddig nincs probléma, amíg `f()` nem változtat `a` értékén (amit ugyebár mutatón keresztül adunk át).

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```
$(\forall x \exists y ((x<y)\wedge(y \text{ prím})))$

$(\forall x \exists y ((x<y)\wedge(y \text{ prím}))\wedge(SSy \text{ prím})) ↩
   )$

$(\exists y \forall x (x \text{ prím}) \supset (x<y)) $

$(\exists y \forall x (y<x) \supset \neg (x \text{ prím}))$
```

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/MatLog_LaTeX

Megoldás videó: https://youtu.be/ZexiPy3ZxsA, https://youtu.be/AJSXOQFF_wk

Tanulságok, tapasztalatok, magyarázat...

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

1. egész

```
int a;
```

2. egészre mutató mutató

```
int *pa;
```

3. egész referenciája

```
int &ra = a; //Kötelező inicializálni, üres referencia nincs
```

4. egészek tömbje

```
int arr[N]; //ha N nem konstans, akkor VLA-t kapunk (csak C99-ben, de  ↩
   sok fordító C++-ban is elfogadja (nem standard!))
```

5. egészek tömbjének referenciája (nem az első elemé)

```
int (&rarr)[N] = arr;
```

6. egészre mutató mutatók tömbje

```
int *parr[N];
```

7. egészre mutató mutatót visszaadó függvény

```
int *func();
```

8. egészre mutató mutatót visszaadó függvényre mutató mutató

```
int *(*funcp)() = func;
```

9. egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény

```
int (*func2(int))(int, int);
```

10. függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

```
int (*(*func2p)(int))(int, int);
```

Mit vezetnek be a programba a következő nevek?

1.
```
int a;
```
a egy egész

2.
```
int *b = &a;
```
b egy egészre mutató mutató ami a-ra mutat

3.
```
int &r = a;
```
r egy referencia egy egészre (a-ra)

4.
```
int c[5];
```

c egy öt egészből álló tömb

5.
```
int (&tr)[5] = c;
```
tr egy referencia egy öt egészből álló tömbre (c-re)

6.
```
int *d[5];
```
d egészre mutató mutatók 5 elemű tömbje

7.
```
int *h();
```
h egy egészre mutató mutatót visszaadó függvény (nincs argumentuma)

8.
```
int *(*l)();
```
l egy függvénymutató, ami egy egészre mutató mutatót visszaadó argumentum nélküli függvényre mutat

9.
```
int (*v(int c))(int a, int b)
```
v egy függvény, aminek egy egész argmentuma van, és egy függvénymutatót ad vissza egy függvényre, aminek két egész argumentuma van (ezek meg vannak nevezve, ami mindössze a dokumentálás segítésére szolgál, ugyanúgy, mint függvényprototípusoknál), és egészet ad vissza

10.
```
int (*(*z)(int))(int, int);
```
z egy függvénymutató, ami egy olyan függvényre mutat, melynek egy egész argmentuma van, és egy függvénymutatót ad vissza egy függvényre, aminek két egész argumentuma van, és egészet ad vissza

Megoldás videó: BHAX 357. adás.

Az utolsó két deklarációs példa demonstrálására két olyan kódot írtunk, amelyek összahasonlítása azt mutatja meg, hogy miért érdemes a **typedef** használata: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr.c, bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr2.c.

```
#include <stdio.h>

int
sum (int a, int b)
{
    return a + b;
}

int
mul (int a, int b)
{
    return a * b;
}

int (*sumormul (int c)) (int a, int b)
{
    if (c)
        return mul;
    else
        return sum;

}

int
main ()
{

    int (*f) (int, int);

    f = sum;

    printf ("%d\n", f (2, 3));

    int (*(*g) (int)) (int, int);

    g = sumormul;

    f = *g (42);

    printf ("%d\n", f (2, 3));

    return 0;
}
```

```
#include <stdio.h>

typedef int (*F) (int, int);
typedef int (*(*G) (int)) (int, int);
```

```
int
sum (int a, int b)
{
    return a + b;
}

int
mul (int a, int b)
{
    return a * b;
}

F sumormul (int c)
{
    if (c)
        return mul;
    else
        return sum;
}

int
main ()
{

    F f = sum;

    printf ("%d\n", f (2, 3));

    G g = sumormul;

    f = *g (42);

    printf ("%d\n", f (2, 3));

    return 0;
}
```

3.9. Vörös Pipacs Pokol/csiga diszkrét mozgási parancsokkal

Megoldás videó: https://youtu.be/Fc33ByQ6mh8

Megoldás forrása: https://github.com/nbatfai/RedFlowerHell

4. fejezet

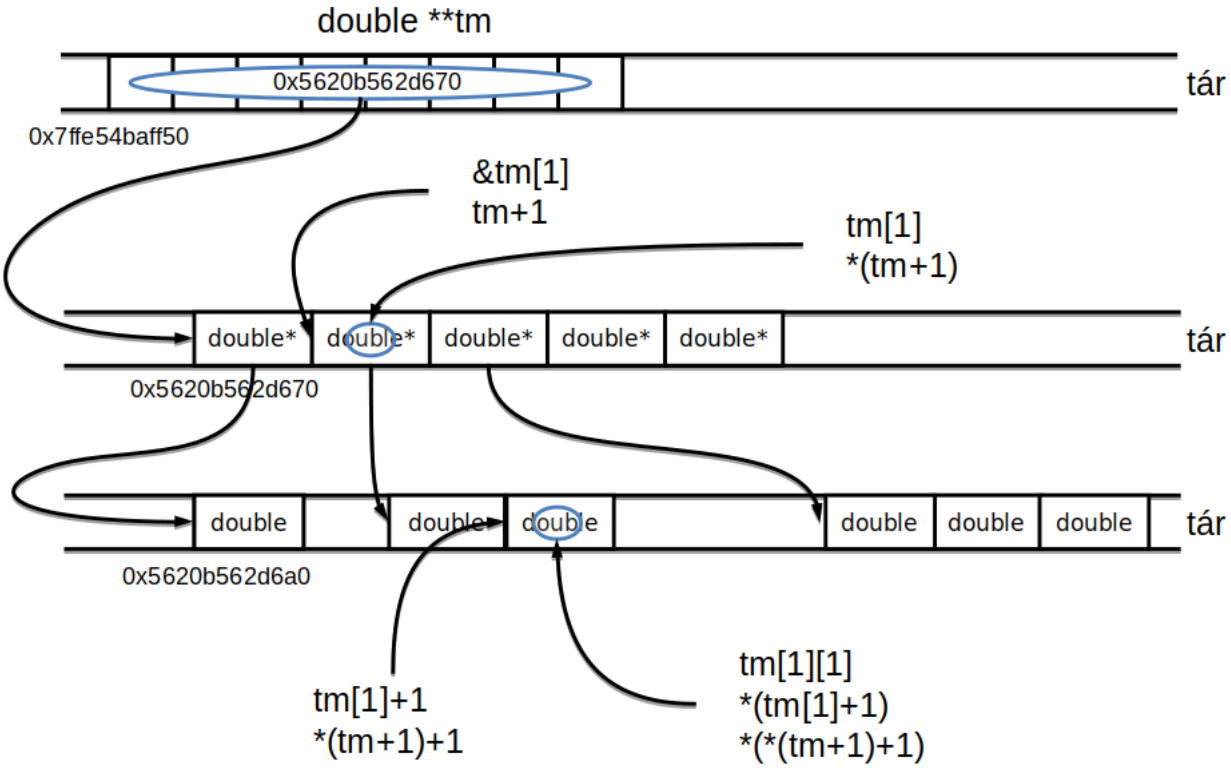
Helló, Caesar!

4.1. double ** háromszögmátrix

Írj egy olyan `malloc` és `free` párost használó C programot, amely helyet foglal egy alsó háromszög mátrixnak a szabad tárban!

Megoldás videó: https://youtu.be/1MRTuKwRsB0, https://youtu.be/RKbX5-EWpzA.

Megoldás forrása: `bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c`



4.1. ábra. A double ** háromszögmátrix a memóriában

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

A megoldásomban definiáltam egy struktúrát annak érdekében, hogy az alsó háromszögmátrixunk egy jól behatárolható típussal rendelkezzen. Ez a nyers `double**` mutatókhoz képest plusz szemantikai jelentést hordoz. Ha mondjuk egy tényleges mátrixokkal dolgozó könyvtárat akarunk létrehozni C-ben, nem érdemes a könyvtár használójának nyers mutatókat visszaadni, hiszen ez rengeteg hibalehetőséghez vezet. Ehhez hasonló megoldások nagyon gyakran jelentkeznek C-ben, és nem nehéz látni, hogy ez hogy vezetett el az osztályok elterjedéséhez. Az ilyen mintázatokat (csinálunk egy struktúrát és függvényeket, melyek ezzel dolgoznak) nevezhetnénk a "szegényember osztályainak". (Az LZW feladat esete hasonló.)

Ezzel a példával jól lehet szemléltetni a templatek értelmét is, hiszen így, ilyen formában csak double értékeket tudunk az alsó háromszögmátrixunkban tárolni. Mi van akkor, ha komplex számokat szeretnénk használni? A C nyelv határán vagyunk, ugyanis C-ben erre a problémára csak igen kellemetlen megoldások léteznek.

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

//Alsó háromszögmátrix (lower triangular matrix) struktúra
typedef struct {int size; double** data;} l_tri_mat;

l_tri_mat ltm_create_matrix(int size);
void ltm_delete_matrix(l_tri_mat* matrix);

double ltm_get_value(l_tri_mat matrix, int row, int col);
int ltm_set_value(l_tri_mat matrix, int row, int col, double value);

void ltm_print_matrix(l_tri_mat matrix, int precision);


double randd(double min, double max);




int main()
{
    srand(42); //Csak azért, hogy legyen mivel feltöltenünk a mátrixot

    l_tri_mat mat = ltm_create_matrix(16);

    for(int i = 0; i < mat.size; i++)
        for(int j = 0; j < mat.size; j++) //Próbaként írjunk a mátrix felső részébe is (nem lesz tárolva)
            ltm_set_value(mat, i, j, randd(-10, 10));

    ltm_print_matrix(mat, 2);
```

```
    printf("mat.data:\t0x%lX\nmat.data[0]:\t0x%lX\n*mat.data[0]:\t%lf\n",  ↩
       mat.data, mat.data[0], *mat.data[0]);

    ltm_delete_matrix(&mat);

    return 0;
}


//Lefoglal egy size méretű háromszögmátrixot
l_tri_mat ltm_create_matrix(int size)
{
    l_tri_mat result;
    result.size = size;

    result.data = malloc(sizeof(double*) * size);

    for(int i = 0; i < size; i++)
        result.data[i] = calloc(i + 1, sizeof(double));

    return result;
}

//Törli a megadott a.h. mátrixot, és felszabadítja az annak lefoglalt  ↩
   helyet
void ltm_delete_matrix(l_tri_mat* matrix)
{
    for(int i = 0; i < matrix->size; i++) free(matrix->data[i]);

    free(matrix->data);
    matrix->data = matrix->size = 0;
}

//Visszaadja a mátrix row-adik sorában és col-adik oszlopában szereplő  ↩
   értéket (vagy 0-át ha nem a határokon kívül vagyunk)
double ltm_get_value(l_tri_mat matrix, int row, int col)
{
    if(row < 0 || col < 0 || row >= matrix.size || col > row) return 0.0;

    return matrix.data[row][col];
}

//Beállítja a megfelelő helyen lévő értéket a mátrixban (ha row és col  ↩
   határokon kívül vannak, 0-ával tér vissza)
int ltm_set_value(l_tri_mat matrix, int row, int col, double value)
{
    if(row < 0 || col < 0 || row >= matrix.size || col > row) return 0;

    matrix.data[row][col] = value;
```

```
    return 1;
}

//Alsó háromszögmátrix kiírása
void ltm_print_matrix(l_tri_mat matrix, int precision)
{
    for(int i = 0; i < matrix.size; i++)
    {
        for(int j = 0; j < matrix.size; j++)
            printf("% .*f\t", precision, ltm_get_value(matrix, i, j));

        printf("\n");
    }
}


//Random double érték min és max között
double randd(double min, double max)
{
    return min + (double)rand() / RAND_MAX * (max - min);
}
```

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás forrása: egy részletes feldolgozása az posztban, lásd az e.c forrást.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

Nincs nehéz dolgunk, az XOR titkosítás kb. a legkönnyebb titkosítási módszer. Egyedül arra kell figyelnünk, hogy itt mindenképpen puffereket kell használnunk, nem hagyatkozhatunk a \0 záró karakterre, mivel a titkosított kódban bárhol előfordulhatnak null karakterek. (Ezt el lehet kerülni pl. base64 kódolással) Itt egy egyszerű implementáció:

```
void xor_encrypt(char *key, char *buffer, int buffer_size)
{
    int keysize = strlen(key);

    for(int i = 0; i < buffer_size; i++)
        buffer[i] ^= key[i % keysize];
}
```

A program többi része lényegében csak beolvasás és kiírás:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void xor_encrypt(char *key, char *buffer, int buffer_size);

#define MAX_KEY_LEN 20
#define BUFFER_SIZE 256

int main(int argc, char **argv)
{
    char key[MAX_KEY_LEN], buffer[BUFFER_SIZE];

    if(argc > 1) strncpy(key, argv[1], MAX_KEY_LEN);
    else
    {
        printf("Encryption key: ");
        fgets(key, MAX_KEY_LEN, stdin);
    }

    int n_bytes;
    while(n_bytes=read(0, buffer, BUFFER_SIZE))
    {
        xor_encrypt(key, buffer, n_bytes);
        write(1, buffer, n_bytes);
    }


    return 0;
}
```

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#exor_titkosito

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

4.4. C EXOR törő

Írj egy olyan C programot, amely megtöri az első feladatban előállított titkos szövegeket!

Megoldás forrása: egy részletes feldolgozása az posztban, lásd az t.c forrást.

Hmm... 8 egymásba ágyazott for ciklus? Sőt, n egymásba ágyazott for ciklus, a keresett kulcs hossza alapján... Kell lennie jobb megoldásnak. A megoldáshoz jó kiindulópont lehet, ha a kulcsokra afféle számokként gondolunk, a megengedett karakterek számának megfelelő számrendszerben. z alábbi két fügvénnyel végig tudunk lépegetni az összes `min_len` és `max_len` közti hosszúságú `allowed_ch` karakterekből álló kulcson:

```
void first_key(char *allowed_ch, int min_len, char *result)
{
    for(int i = 0; i < min_len; i++) result[i] = allowed_ch[0];
    result[min_len] = '\0';
}

int next_key(char *allowed_ch, int max_len, char *key)
{
    int carry = 1, allowed_ch_len = strlen(allowed_ch), key_len = strlen( ↩
       key);

    while(carry)
    {
        if(*key == '\0')
        {
            if(key_len == max_len) return 0;
            else
            {
                key[0] = allowed_ch[0];
                key[1] = '\0';
            }
        }

        int pos;
        for(pos = 0; pos < allowed_ch_len; pos++)
            if(allowed_ch[pos] == *key) break;

        if(pos == allowed_ch_len) return -1;
        pos = (pos + 1) % allowed_ch_len;

        *key = allowed_ch[pos];
        if(pos != 0) carry = 0;
        key++;
    }

    return 1;
}
```

Mivel nem használjuk egyszerre az összes lehetőséget, elég egy kinevezett első érték, és az adott kulcsnak a rákövetkezője. (forward iteration) Ennyivel már könnyedén (és hatékonyan) megírhatjuk a brute-force programunk fő ciklusát.

```
first_key(allowed_chars, min_key_size, key);
do
{
    xor_encrypt(key, buffer, n_bytes);

    if(heuristic(buffer, n_bytes))
    {
        printf("%s\n\n", buffer);
    }

    xor_encrypt(key, buffer, n_bytes);
} while(next_key(allowed_chars, max_key_size, key) > 0);
```

4.5. Neurális OR, AND és EXOR kapu

R

Megoldás videó: https://youtu.be/Koyw6IH5ScQ

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

4.6. Hiba-visszaterjesztéses perceptron

Fontos, hogy ebben a feladatban még nem a neurális paradigma megismerése a cél, hanem a többrétegű perceptron memóriakezelése (lásd majd a változó argumentumszámú konstruktorban a double *** szerkezetet).

Megoldás videó: https://youtu.be/XpBnR31BRJY

Megoldás forrása: https://github.com/nbatfai/nahshon/blob/master/ql.hpp#L64

Lépjünk vissza egy kicsit... C++-ban vagyunk, így megtehetjük, és miért is ne tennénk, hogy amennyiben nem muszáj, nem kínlódunk magunk a memóriakezeléssel. Ebben az esetben céljainknak tökéletesen megfelel az `std::vector<>`, hiszen közvetlen elérést biztosít, futásidőben meghatározott méretű, és megfelelő paraméterekkel nem használ több tárhelyet a szükségesnél.

```
typedef std::pair<double, std::vector<double>> neuron;
typedef std::vector<neuron> layer;

std::vector<layer> neurons;
```

```
//Kibontva:
std::vector<std::vector<std::pair<double, std::vector<double>>>> neurons;
```

Habár lehet, hogy túlzásnak tűnik a `neurons` deklaráció kibontása, de nincs számottevő teljesítménybeli különbség több dimenzió esetén sem... Ha teljesen pontosak akarunk lenni, le lehet tesztelni és össze lehet hasonlítani pár különböző implementáció sebességét, de a vektor esetében ezt már rengetegen megtették. Főleg, mivel itt nem változtatjuk a vektoraink méretét, nem kell számítanunk a teljesítmény romlására.

Az adatmodellünk megvan, következő lépésként építsünk köré egy osztályt. Ez lehet egy nagyon könnyű feladat is, de lehet igen nehéz is, alaposságtól függően. Ha egy jól megtervezett osztályt akarunk készíteni, amit bárki kezébe adhatunk, sok dologra oda kell figyelnünk:

- elérés-módosítok (access modifiers): public, protected, private
- másoló, mozgató konstruktorok
- másoló, mozgató értékadó operátorok
- elérő és módosító függvények
- referenciák használata (a felesleges másolások elkerülése érdekében)
- amit nem változtatunk, az legyen konstans (a függvények végén akkor van const, ha az nem változtatja meg a példány állapotát) (const-correctness)

Alább található az ezek alapján elkészített megoldásom. (Csak az osztály szerkezete, maga az implementáció még egy következő lépés.)

```
class neural_net
{
public:
    neural_net(const std::vector<unsigned int>& topology = std::vector< ↩
       unsigned int>());
    neural_net(const neural_net& other);
    neural_net(neural_net&& other);
    ~neural_net();

    neural_net& operator=(const neural_net& other);
    neural_net& operator=(neural_net&& other);

    std::vector<double> input() const;
    std::vector<double>::const_iterator input_begin() const; //Iterátorok  ↩
       arra az esetre, ha nincs szükségünk másolásra
    std::vector<double>::const_iterator input_end() const;
    void feed_input(const std::vector<double>& input);

    std::vector<double> output() const;
```

```
    std::vector<double>::const_iterator output_begin() const;
    std::vector<double>::const_iterator output_end() const;
    void back_propagate(const std::vector<double>& output);

private:
    /* Példa topológára: 3 4 3 2
       Értelmezés: 3+1 bemeneti neuron, egy rejtett réteg 4+1 neuronnal,
                   mégegy rejtett réteg 3+1 neuronnal, 2 kimeneti neuron
                   (a +1 neuronok az eltolósúly-értékekhez kellenek (bias))
    */
    std::vector<unsigned int> topology;

protected:
    /* std::pair-t használunk:
       számított érték/bemeneti érték + a megfelelő súlyok vektora (a ↩
          következő réteg elemszámától függ)
       Lehetne osztály is, bizonyos esetben előnyös lehet, de maradjunk egy ↩
           egyszerűbb leképezésnél. */
    typedef std::pair<double, std::vector<double>> neuron;
    typedef std::vector<neuron> layer;

    std::vector<layer> neurons;

    double neuron_value(int layer, int neuron) const;
    void set_neuron_value(int layer, int neuron, double value);

    double weight_value(int layer_from, int neuron_from, int neuron_to) ↩
       const;
    void set_weight_value(int layer_from, int neuron_from, int neuron_to, ↩
       double weight);
};
```

4.7. Vörös Pipacs Pokol/írd ki, mit lát Steve

Megoldás videó: https://youtu.be/-GX8dzGqTdM

Megoldás forrása: https://github.com/nbatfai/RedFlowerHell

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

Megoldás videó: https://youtu.be/gvaqijHlRUs

Megoldás forrása: `bhax/attention_raising/CUDA/mandelpngt.c++` nevű állománya.

A Mandelbrot halmazt 1980-ban találta meg Benoit Mandelbrot a komplex számsíkon. Komplex számok azok a számok, amelyek körében válaszolni lehet az olyan egyébként értelmezhetetlen kérdésekre, hogy melyik az a két szám, amelyet összeszorozva -9-et kapunk, mert ez a szám például a 3i komplex szám.

A Mandelbrot halmazt úgy láthatjuk meg, hogy a sík origója középpontú 4 oldalhosszúságú négyzetbe lefektetünk egy, mondjuk 800x800-as rácsot és kiszámoljuk, hogy a rács pontjai mely komplex számoknak felelnek meg. A rács minden pontját megvizsgáljuk a $z_{n+1}=z_n{}^2+c$, (0<=n) képlet alapján úgy, hogy a c az éppen vizsgált rácspont. A z0 az origó. Alkalmazva a képletet a

- $z_0 = 0$
- $z_1 = 0^2+c = c$
- $z_2 = c^2+c$
- $z_3 = (c^2+c)^2+c$
- $z_4 = ((c^2+c)^2+c)^2+c$
- ... s így tovább.

Azaz kiindulunk az origóból (z_0) és elugrunk a rács első pontjába a z_1 = c-be, aztán a c-től függően a további z-kbe. Ha ez az utazás kivezet a 2 sugarú körből, akkor azt mondjuk, hogy az a vizsgált rácspont nem a Mandelbrot halmaz eleme. Nyilván nem tudunk végtelen sok z-t megvizsgálni, ezért csak véges sok z elemet nézünk meg minden rácsponthoz. Ha eközben nem lép ki a körből, akkor feketére színezzük, hogy az a c rácspont a halmaz része. (Színes meg úgy lesz a kép, hogy változatosan színezzük, például minél későbbi z-nél lép ki a körből, annál sötétebbre).

***

A feladat eredetileg C programot kér, amitől azonban a tényleges megoldás eltért. Érthető ez, mert a C-ben elérhető `libpng` igen alacsony szintű könyvtár, esetünkben elég kellemetlen lenne használni. Kíváncsiságból azért utánajártam, hogy hogy is néz ki a PNG formátum, szép feladatot lehetne belőle csinálni, hogy hogyan is lehet egyszerű PNG képeket építeni bájtról bájtra.

Ám ez az eredeti feladattól egészen messzire vinne, így végül elvetettem a "találjuk fel újra a PNG-t" ötletet. Egy kevés keresgélés után viszont találtam nem is egy könnyen implementálható képfájl-formátumot; olyanokat, amik pont arra jók, hogy nyers színadatokat dobáljunk bele (hiszen tömörítésre, checksumokra, stb. most nincs szükségünk). Ezek közül a Netpbm-et találtam a legmegfelelőbbnek, azon belül is a `.pgm` (Portable GreyMap) típust. Csináltam tehát az ilyen fájlok írásához egy egyszerű implementációt:

```
typedef struct
{
    FILE *file;
    coord size;
    unsigned int pos;
} pgm_write_info;

pgm_write_info pgm_write_hdr(FILE *f, coord img_size)
{
    pgm_write_info info = {f, img_size, 0};

    if(!fprintf(f, "P5 %u %u 255\n", img_size.x, img_size.y))
        info.file = NULL;

    return info;
}

int pgm_write_pixel(pgm_write_info *i, unsigned char pixel)
{
    if(i->pos >= i->size.x * i->size.y) return -1;
    i->pos++;

    return fwrite(&pixel, sizeof(unsigned char), 1, i->file);
}
```

Kisebb kitérő után tehát meglett a megoldás egyszerű képfájlok létrehozására. Innentől már csak számolni kellett egy keveset, és meg is jelent a Mandelbrot-halmaz a szemeim előtt.

A Mandelbrot iteráció függvénye:

```
//Nullával térünk vissza, ha c a Mandelbrot-halmaz eleme, egyébként egy ↩
   színezéshez használható 1-255 közötti értéket adunk
unsigned char mandelbrot_iter(complexd c, unsigned int max_iter)
```

```
{
    complexd z = c;
    unsigned char result = 0;

    for(unsigned int i = 1; i <= max_iter; i++)
    {
        double re2 = z.re * z.re;
        double im2 = z.im * z.im;
        if(re2 + im2 > 4)
        {
            result = 1 + i * 254 / max_iter;
            break;
        }

        //(a+bi)^2 + c + di = a*a-b*b+c + (2*a*b+d)*i
        z.im = 2 * z.re * z.im + c.im;
        z.re = re2 - im2 + c.re;
    }

    return result;
}
```

A kép pixelein végigfutó ciklus:

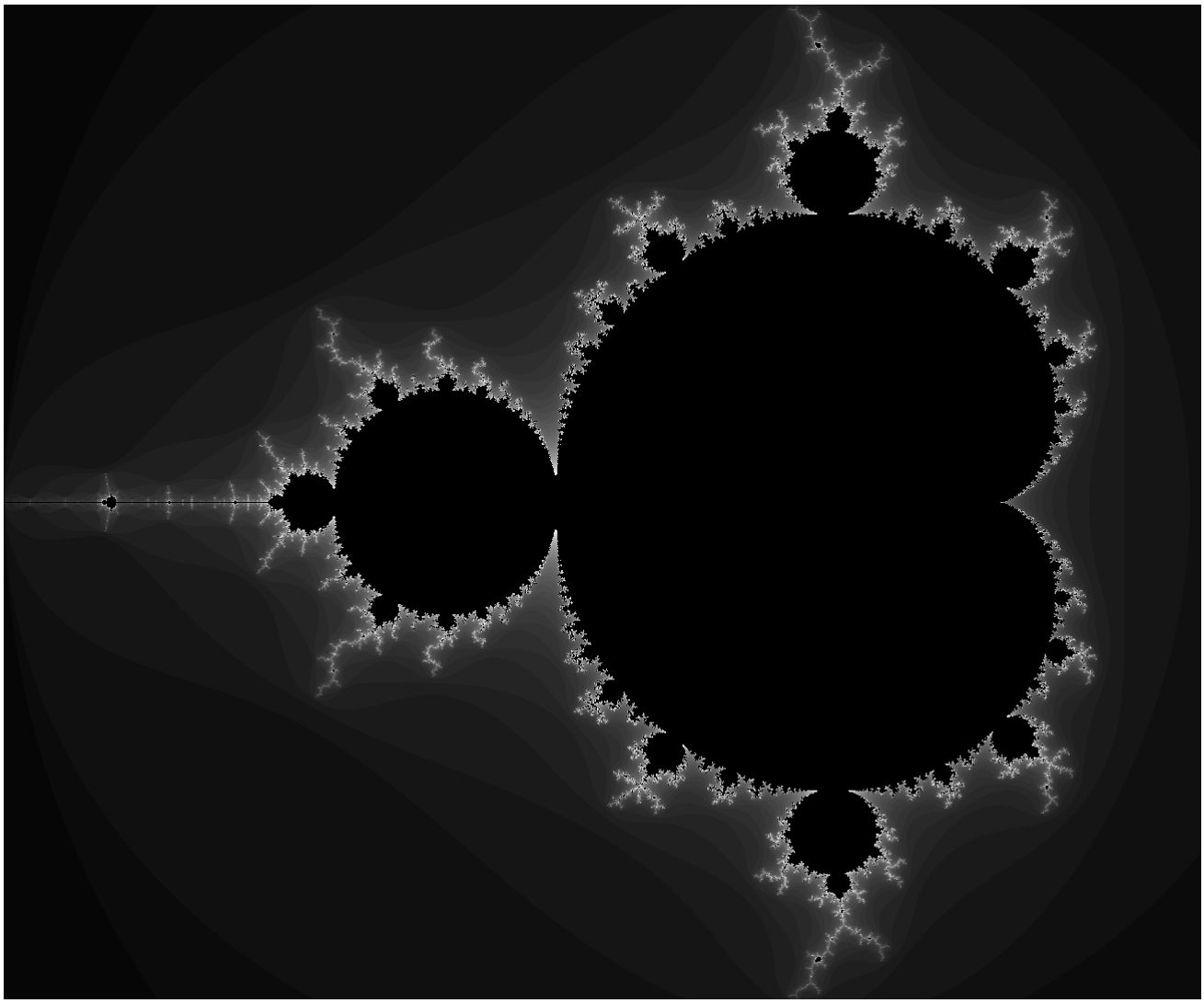
```
for(coord pos = {0, 0}; pos.y < img_size.y; pos.y++)
    for(pos.x = 0; pos.x < img_size.x; pos.x++)
    {
        unsigned char i = mandelbrot_iter(
            img_coords_to_complex(pos, img_size, top_left, bottom_right),
            max_iter);

        pgm_write_pixel(&pgm_inf, i);
    }
```

A kép koordinátái és a komplex sík pontjai közti átmenethez szükséges függvények:

```
typedef struct {double re, im;} complexd;
typedef struct {unsigned int x, y;} coord;

double lerpd(double a, double b, double t)
{
    return a * (1 - t) + b * t;
}
```

```
complexd img_coords_to_complex(coord point, coord img_size, complexd  ↩
   top_left, complexd bottom_right)
{
    complexd result;
    result.re = lerpd(top_left.re, bottom_right.re, (double)point.x /  ↩
       img_size.x);
    result.im = lerpd(top_left.im, bottom_right.im, (double)point.y /  ↩
       img_size.y);

    return result;
}
```

És az eredmény:



5.1. ábra. A Mandelbrot halmaz a komplex síkon

5.2. A Mandelbrot halmaz a `std::complex` osztállyal

Írj olyan C++ programot, amely kiszámolja a Mandelbrot halmazt!

Megoldás videó: https://youtu.be/gvaqijHlRUs

Megoldás forrása: BHAX repó, https://gitlab.com/nbatfai/bhax/-/blob/master/attention_raising/Mandelbrot/-3.1.2.cpp

A Mandelbrot halmaz pontban vázolt ismert algoritmust valósítja meg a repó `bhax/attention_raising/Mandelbrot/3.1.2.cpp` nevű állománya.

Az `std::complex` használata megkönnyíti a komplex számolásokat:

```
unsigned char mandelbrot_iter(std::complex<double> c, unsigned int max_iter ↩
    )
{
    std::complex<double> z = c;
    unsigned char result = 0;

    for(unsigned int i = 1; i <= max_iter; i++)
    {
        if(std::abs(z) > 2)
        {
            result = 1 + i * 254 / max_iter;
            break;
        }

        z = z * z + c;
    }

    return result;
}
```

Emellett, mivel C++-ban vagyunk, a képfájl írása is könnyebbé válik (`std::ofstream`).

5.3. Biomorfok

Megoldás videó: https://youtu.be/IJMbgRzY76E

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Biomorf

A biomorfokra (a Julia halmazokat rajzoló bug-os programjával) rátaláló Clifford Pickover azt hitte természeti törvényre bukkant: https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf (lásd a 2307. oldal aljától).

A különbség a Mandelbrot halmaz és a Julia halmazok között az, hogy a komplex iterációban az előbbiben a c változó, utóbbiban pedig állandó. A következő Mandelbrot csipet azt mutatja, hogy a c befutja a vizsgált összes rácspontot.

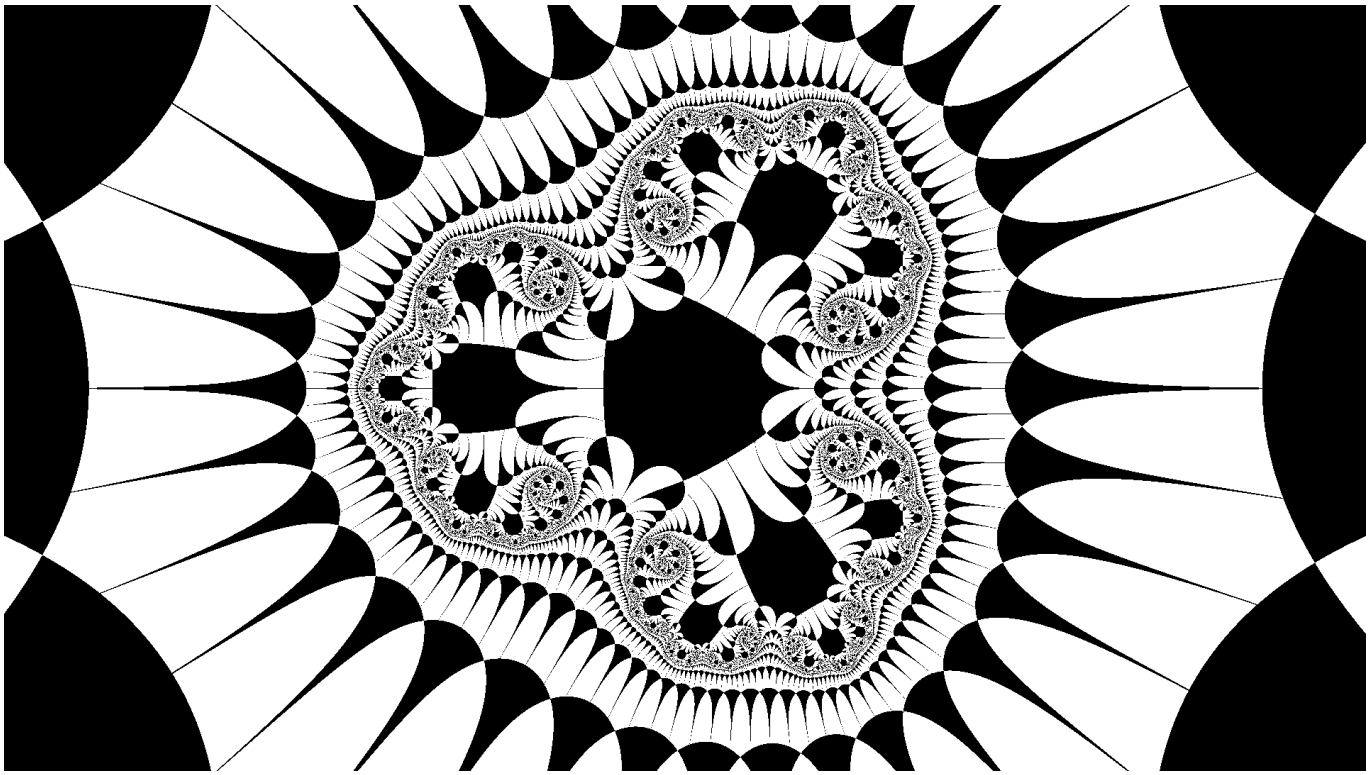
```
// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
  {
    for ( int k = 0; k < szelesseg; ++k )
      {

        // c = (reC, imC) a halo racspontjainak
        // megfelelo komplex szam

        reC = a + k * dx;
        imC = d - j * dy;
        std::complex<double> c ( reC, imC );

        std::complex<double> z_n ( 0, 0 );
        iteracio = 0;

        while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
          {
            z_n = z_n * z_n + c;

            ++iteracio;
          }
```

Ezzel szemben a Julia halmazos csipetben a cc nem változik, hanem minden vizsgált z rácspontra ugyanaz.

```
  // j megy a sorokon
  for ( int j = 0; j < magassag; ++j )
  {
      // k megy az oszlopokon
      for ( int k = 0; k < szelesseg; ++k )
      {
          double reZ = a + k * dx;
          double imZ = d - j * dy;
          std::complex<double> z_n ( reZ, imZ );

          int iteracio = 0;
          for (int i=0; i < iteraciosHatar; ++i)
          {
              z_n = std::pow(z_n, 3) + cc;
              if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
              {
                  iteracio = i;
                  break;
              }
          }
```

A bimorfos algoritmus pontos megismeréséhez ezt a cikket javasoljuk: https://www.emis.de/journals/-TJNSA/includes/files/articles/Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf. Az is jó gyakorlat, ha magából ebből a cikkből from scratch kódoljuk be a sajátunkat, de mi a királyi úton járva a korábbi Mandelbrot halmazt kiszámoló forrásunkat módosítjuk. Itt van tehát a módosított iterációs függvény: (Ez a változat a $z = z^3 + 0.5$ biomorfot számolja)

```
unsigned char biomorph_iter(complexd z, complexd c, unsigned int max_iter)
{
    for(unsigned int i = 1; i <= max_iter; i++)
    {
        if(z.re < -10 || z.re > 10 || z.im < -10 || z.im > 10)
            break;

        complexd z_next = {z.re * (z.re*z.re - 3*z.im*z.im) + c.re, z.im *  ↩
           (3*z.re*z.re - z.im*z.im) + c.im};
        z = z_next;
    }

    if((z.re > -10 && z.re < 10) || (z.im > -10 && z.im < 10))
        return 0;

    return 255;
}
```



5.2. ábra. A Radiolarian névre hallgató biomorf

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó: https://youtu.be/gvaqijHlRUs

Megoldás forrása: `bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu` nevű állománya.

5.5. Mandelbrot nagyító és utazó C++ nyelven

Építs GUI-t a Mandelbrot algoritmusra, lehessen egérrel nagyítani egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteréció bejárta z_n komplex számokat!

Megoldás videó: Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a_mandelbrot_halmazzal.

Megoldás forrása: az ötödik előadás 26-33 fólia, illetve https://sourceforge.net/p/udprog/code/ci/master/-tree/source/binom/Batfai-Barki/frak/.

Az előző megoldások kódját tök jól felhasználhatjuk, csak a rajzolással kapcsolatos kódot össze kell hoznunk egy grafikus felülettel. Qt-ben viszonylag könnyű dolgozni, talán nagy részben azért, mert maga a könyvtár nagyon jól dokumentált, és mert a felhalsználói interfészeknél megszokott esemény-alapú architektúrát követő signal-slot mechanizmussal könnyű elérni az éppen igényeinknek megfelelő viselkedést.

Csak összehasonlításként, egyszerűbb grafikus felületek építésével próbálkoztam már korábban Windows API-ban is, és noha jó dolog mélyen belelátni a windows shell működésébe, de mindenesetre hálát tudok

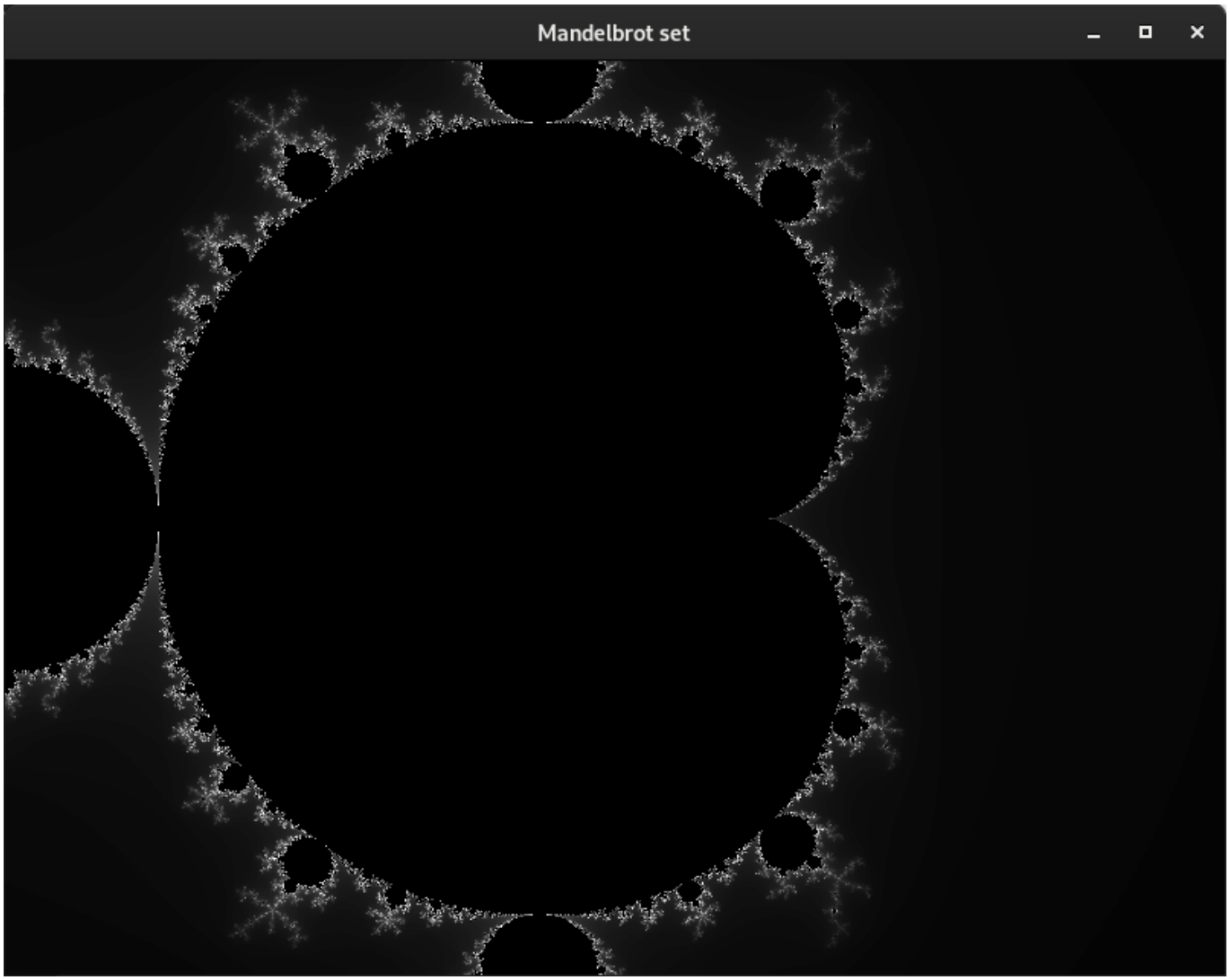
adni mindazoknak a fejlesztőknek, akik ezeket a natív könyvtárakat becsomagolták lényegesen könnyebben kezelhető (ráadásul legtöbb esetben cross-platform) toolkitekbe.

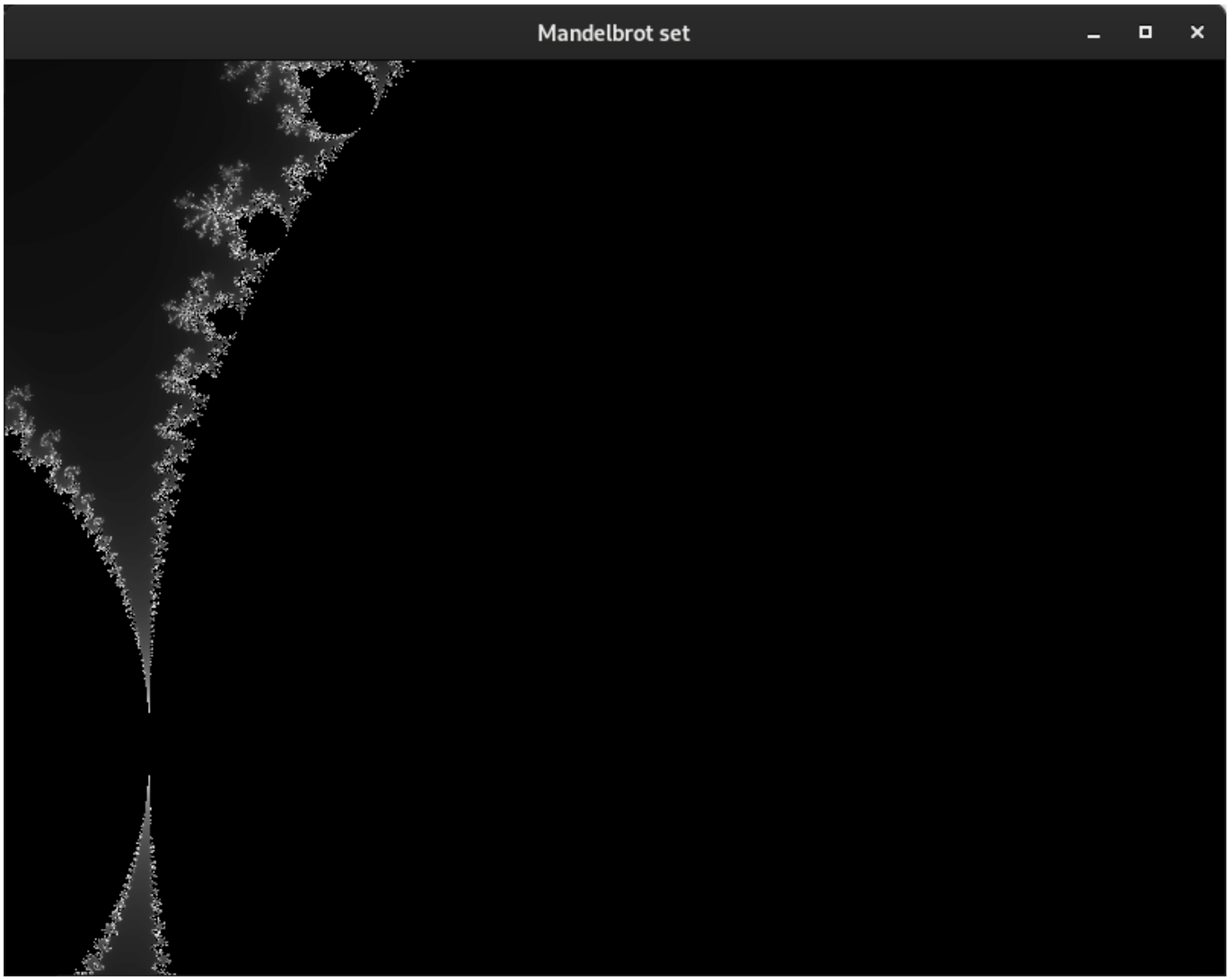
A Mandelbrot-halmazzal kapcsolatos rész annyiban változott, hogy most képfájl helyett egy `QImage`-be rajzoltunk, melyet aztán az ablakunk `paintEvent()`-jében meg is jelenítettünk.

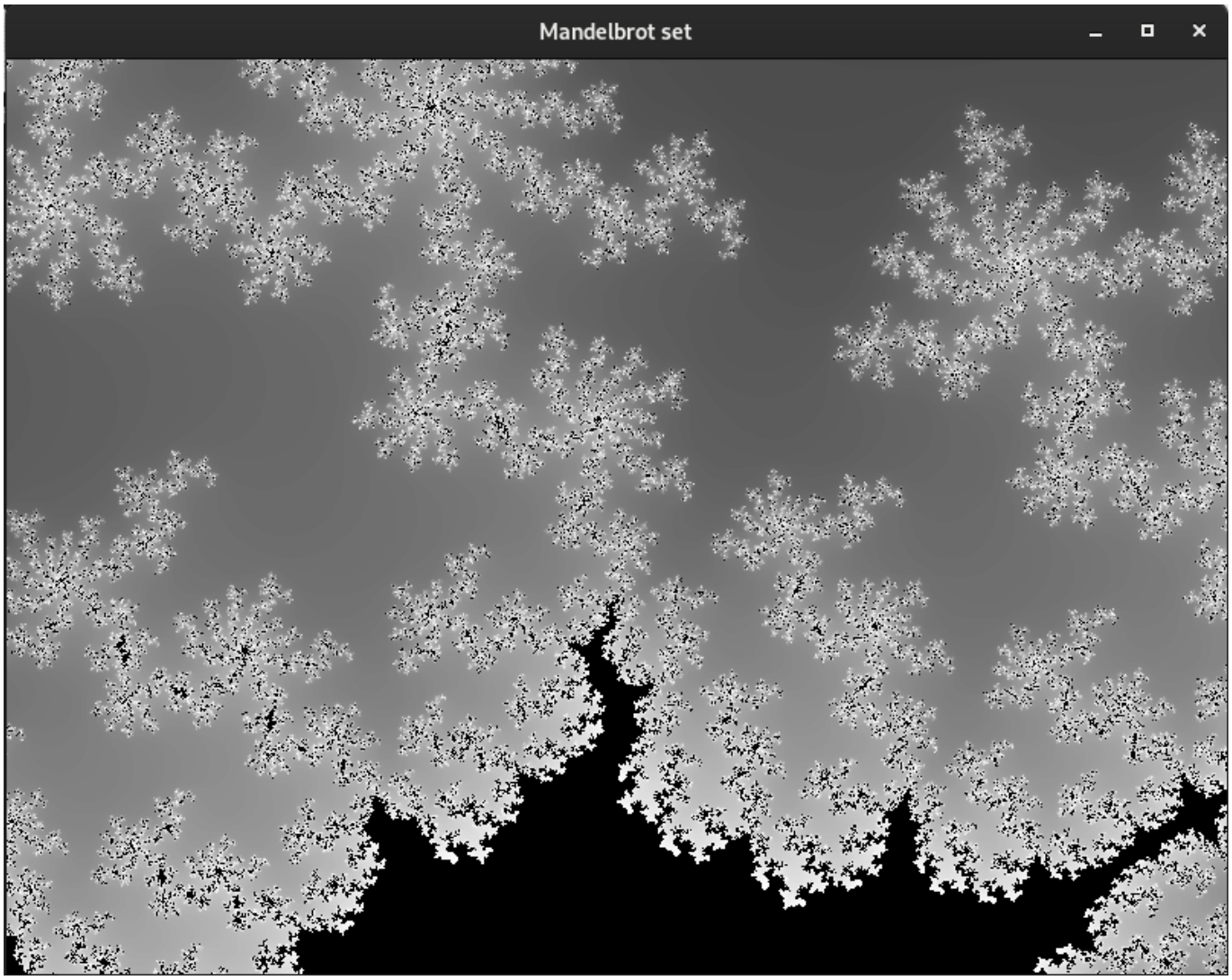
A panning/zooming (mozgatás, nagyítás) hasonlóan működik, mint egy szokásos képmegjelenítő programban, tehát nagyításnál/kicsinyítésnél az egérmutató adja a fix pontot, mozgatásnál meg a kép követi az kurzor mozgását. Annyiban nehezebb ennél a feladat, hogy a képet időnként újra is kell számolnunk. Namost, minden egéreseménynél nem érdemes ezt megtenni, mert a kapott program nagyon nem lenne reszponzív, és rengeteg számítás szinte rögtön feleslegbe menne. Így egy időzítőt felhasználva, a programom csak akkor számol, ha egy megadott időköz eltelt új esemény nélkül. Ezt ugyan még mindig a fő szálon végzi a nehéz munkát (egy következő lépés lenne a dolgozó szálak használata), de nagyjából csak akkor, amikor szükséges, amikor nincs további pan/zoom esemény.

További technikai részlet, hogy az ablak átméretezésénél mind a háttér `QImage`-ünket, mind a jelenlegi nézet paramétereit is frissítenünk kell. Ez a felhasználói felületek létrehozásának talán legkellemetlenebb, és legtöbb megfontolást igénylő része. Jelen esetben csak egy képet jelenítünk meg, de ha tényleges vezérlőket használunk, akkor sajnos sok esetben az általunk megálmodott layout csak egy kezdet. Minden értelmes keretek közti ablakméretre gondolnunk kell.

Egy pár screenshot:


Mandelbrot set


Mandelbrot set


Mandelbrot set

A legutolsó annak a példája, hogy mi történik, ha túl messzire zoomolunk, méghozzá annyira messzire, hogy túlmenjünk a `double` típus által ábrázolható precízión. Ennek a problémának a megoldásához tetszőleges pontosságú arimetikát kéne használnunk, de ez jelenleg a kis projektünk keretein talán túlmutat.

```
#include <QtWidgets>
#include <QtMath>
#include <complex>

typedef std::complex<double> complexd;

template<typename T>
inline T lerp(T a, T b, T t)
{
    return a * (1 - t) + b * t;
}


#define RECALC_DELAY 200
#define SCROLL_SENSITIVITY 0.1
```

```
class MandelbrotWindow : public QWidget
{
private:
    QImage frame;
    QRect view_src_rect;
    complexd c_center, top_left, bottom_right;
    double half_width = 1.0;

    QPoint prev_mouse_pos;

    QTimer recalc_timer;
    QThread recalc_thread;
    QMutex frame_lock;


    void resizeEvent(QResizeEvent *event) override
    {
        QImage new_frame = QImage(width(), height(), QImage:: ↩
           Format_Grayscale8);
        new_frame.fill(0);
        frame = new_frame;
        //frame = frame.scaled(size());
        //view_src_rect = frame.rect();

        update_corner_coords();
        recalc_timer.start();
    }

    void keyPressEvent(QKeyEvent *event) override
    {
        event->ignore();
    }

    void mousePressEvent(QMouseEvent *event) override
    {
        prev_mouse_pos = event->pos();
    }

    void mouseMoveEvent(QMouseEvent *event) override
    {
        if(!(event->buttons() & Qt::LeftButton)) return;

        prev_mouse_pos -= event->pos();
        complexd pan = frame_coords_to_complex(prev_mouse_pos) - top_left;
        c_center += pan;
        view_src_rect.moveTo(view_src_rect.topLeft() + prev_mouse_pos);
        prev_mouse_pos = event->pos();

        update();
```

```
    }

    void mouseReleaseEvent(QMouseEvent *event) override
    {
        update_corner_coords();
        recalculate();
    }

    void wheelEvent(QWheelEvent *event) override
    {
        complexd c_mouse = frame_coords_to_complex(event->pos());
        double scale_factor = qPow(2, (qreal)event->angleDelta().y() /  ↩
           1200); qDebug()<<"wheel "<<event->angleDelta().y()<<" "<< ↩
           scale_factor;

        c_center = (c_center - c_mouse) * scale_factor + c_mouse;
        half_width = half_width * scale_factor;

        view_src_rect.setTopLeft((view_src_rect.topLeft() - event->pos()) * ↩
            scale_factor + event->pos());
        view_src_rect.setBottomRight((view_src_rect.bottomRight() - event-> ↩
           pos()) * scale_factor + event->pos());

        update_corner_coords();
        recalc_timer.start();
        update();
    }

    void paintEvent(QPaintEvent *event) override
    {
        QPainter p(this);

        p.drawImage(rect(), frame, view_src_rect);
    }

    void recalculate()
    {
        qDebug()<<"recalculate\n";
        //frame_lock.lock();

        for(QPoint pt; pt.y() < frame.height(); pt.setY(pt.y() + 1))
        {
            for(pt.setX(0); pt.x() < frame.width(); pt.setX(pt.x() + 1))
            {
                char val = mandelbrot_iter(frame_coords_to_complex(pt),  ↩
                   150);
                frame.setPixel(pt, qRgb(val, val, val));
            }
            view_src_rect = frame.rect();
        }
```

```
    view_src_rect = frame.rect();
    //frame_lock.unlock();

    update();
}

unsigned char mandelbrot_iter(complexd c, unsigned int max_iter)
{
    complexd z = c;
    unsigned char result = 0;

    for(unsigned int i = 1; i <= max_iter; i++)
    {
        double re2 = z.real() * z.real();
        double im2 = z.imag() * z.imag();
        if(re2 + im2 > 4)
        {
            result = 1 + i * 254 / max_iter;
            break;
        }

        //(a+bi)^2 + c + di = a*a-b*b+c + (2*a*b+d)*i
        z.imag(2 * z.real() * z.imag() + c.imag());
        z.real(re2 - im2 + c.real());
    }

    return result;
}

void update_corner_coords()
{
    complexd diag(half_width, half_width * frame.height() / frame.width ↩
        ());
    top_left = c_center - diag;
    bottom_right = c_center + diag;

    qDebug()<<"c_center:     "<<c_center.real()<<"; "<<c_center.imag();
    qDebug()<<"top_left:     "<<top_left.real()<<"; "<<top_left.imag();
    qDebug()<<"bottom_right: "<<bottom_right.real()<<"; "<<bottom_right ↩
        .imag();
}

complexd frame_coords_to_complex(QPoint point)
{
    complexd result;
    result.real(lerp<double>(top_left.real(), bottom_right.real(), ( ↩
        double)point.x() / frame.width()));
    result.imag(lerp<double>(top_left.imag(), bottom_right.imag(), ( ↩
        double)point.y() / frame.height()));
```

```
        return result;
    }

    QPoint complex_coords_to_frame(complexd point)
    {
        point -= top_left;
        complexd b = bottom_right - top_left;
        QPoint result;
        result.setX((int)lerp<double>(0, frame.width(), point.real() / b. ↩
            real()));
        result.setY((int)lerp<double>(0, frame.height(), point.imag() / b. ↩
            imag()));

        return result;
    }

public:
    explicit MandelbrotWindow(QWidget *parent = nullptr)
        : QWidget(parent), recalc_timer(this), recalc_thread(this)
    {
        setWindowTitle("Mandelbrot set");
        resize(800, 600);

        recalc_timer.setSingleShot(true);
        recalc_timer.setInterval(RECALC_DELAY);
        //connect(&recalc_timer, &QTimer::timeout, &recalc_thread, [this]() ↩
            {qDebug()<<"recalc_thread.start();"; recalc_thread.start(); });
        connect(&recalc_timer, &QTimer::timeout, this, [this](){  ↩
            recalculate(); });
        connect(&recalc_thread, &QThread::finished, this, [this](){ update ↩
            (); });
    }
};


int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    MandelbrotWindow window;
    //window.resize(800, 600);
    window.show();

    return app.exec();
}
```

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás videó: https://youtu.be/Ui3B6IJnssY, 4:27-től. Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#id570518

5.7. Vörös Pipacs Pokol/fel a láváig és vissza

Megoldás videó: https://youtu.be/I6n8acZoyoo

Megoldás forrása: https://github.com/nbatfai/RedFlowerHell

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

6. fejezet

Helló, Welch!

6.1. Első osztályom

Valósítsd meg C++-ban és Java-ban a módosított polártranszformációs algoritmust! A matek háttér teljesen irreleváns, csak annyiban érdekes, hogy az algoritmus egy számítása során két normálist számol ki, az egyiket elspájzolod és egy további logikai taggal az osztályban jelzed, hogy van vagy nincs eltéve kiszámolt szám.

Megoldás forrása: a második előadás 17-22 fólia.

Tanulságok, tapasztalatok, magyarázat... térj ki arra is, hogy a JDK forrásaiban a Sun programozói pont úgy csinálták meg ahogyan te is, azaz az OO nemhogy nem nehéz, hanem éppen természetes neked!

Először nézzük meg, hogy nézne ki osztályok nélkül a feladat: (C-ben)

```
double rand_polar()
{
    static double stored;
    static bool hasStored = false;

    if(hasStored)
    {
        hasStored = false;
        return stored;
    }

    double v1, v2, w;

    do
    {
        v1 = (double)rand() / RAND_MAX * 2.0 - 1;
        v2 = (double)rand() / RAND_MAX * 2.0 - 1;
        w = v1 * v1 + v2 * v2;
    } while(w > 1);
```

```
    double r = sqrt(-2.0 * log(w) / w);

    hasStored = true;
    stored = r * v2;

    return r * v1;
}
```

Ez a megoldás azt a kevés állapotot, amit meg kell tartanunk, statikus változókban tárolja. Ehhez képest egy objektum-orientált megközelítésben ezt az állapotot egy osztályba foglaljuk: (C++)

```
class polargen
{
public:
    polargen() = default;

    double next() { /*...*/ }

private:
    double stored;
    bool hasStored;
}
```

Ennek a nem statikus megközelítésnek akkor van igazán értelme, ha pszeudo-random értékeket akarunk generálni, példányokhoz kötött `seed` alapján. Ehhez C++-ban ezesetben ajánlatos az `std::uniform_real_di` osztályt használni egy `std::default_random_engine` generátorral: (összetettebb, mint a `rand()`, de több lehetőséget nyújt)

```
class polargen
{
public:
    polargen()
      : rdev(
            std::chrono::system_clock
               ::now().time_since_epoch().count()
        ),
        r_unif_dist(-1.0, 1.0),
        hasStored(false) {}

    polargen(uint_fast32_t seed)
      : rdev(seed),
        r_unif_dist(-1.0, 1.0),
        hasStored(false) {}

    double next()
    {
        /*...*/
```

```
        //Véletlen számok genrálása a -1.0 és 1.0 közötti értékekre ↩
           beállított std::uniform_real_distribution-nel:
            v1 = r_unif_dist(rdev);
            v2 = r_unif_dist(rdev);

        /*...*/
    }

private:
    std::default_random_engine rdev;
    std::uniform_real_distribution<double> r_unif_dist;
    double stored;
    bool hasStored;
}
```

***

Javában hasonló módon járunk el, viszont itt bele vagyunk kényszerítve egy objektum-orientált keretbe (ez a Java egy gyakran kritizált tulajdonsága).

Kétféle implementáció lehetséges: létrehozhatunk egy statikus osztályt (privát konstuktor, csak statikus elemek) `java.lang.Math.random()`-mal, vagy készíthetünk egy példányosítható osztályt, amely `java.util.Random`-ot használ. Az utóbbi esetében példányonként megadhatunk egy seedet.

Itt a seedelhető változat: (valójában redundáns, mivel lényegében ugyanazt éri el, mint a `Random.nextGaussi`

```
package prog1.welch;

import java.util.Random;

public class polar {
    Random rand;
    boolean hasStoredValue;
    double storedValue;

    public polar() {
        rand = new Random();
        hasStoredValue = false;
    }

    public polar(long seed) {
        rand = new Random(seed);
        hasStoredValue = false;
    }

    double next() {
        if(hasStoredValue) {
            hasStoredValue = false;
            return storedValue;
```

```
        }

        double v1, v2, w;

        do {
            v1 = rand.nextDouble() * 2 - 1;
            v2 = rand.nextDouble() * 2 - 1;
            w = v1 * v1 + v2 * v2;
        } while(w > 1);

        double r = Math.sqrt(-2.0 * Math.log(w) / w);

        hasStoredValue = true;
        storedValue = r * v2;

        return r * v1;
    }
}
```

„ "...térj ki arra is, hogy a JDK forrásaiban a Sun programozói pont úgy csinálták meg ahogyan te is, azaz az OO nemhogy nem nehéz, hanem éppen természetes neked!" ”

Ez majdnem igaz, annyi kiegészítéssel, hogy az előbb említett `java.util.Random` osztály thread-safe, azaz több szálon is használható. (Nem ajánlatos egy `Random` példányt több szál között megosztva használni, de mindenesetre lehetséges.)

6.2. LZW

Valósítsd meg C-ben az LZW algoritmus fa-építését!

Megoldás forrása: https://progpater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egy_pe

Ehhez először hozzunk létre egy bináris fa csomópont struktúrát:

```
typedef struct tag_bintree_node
{
    int data;
    struct tag_bintree_node* left;
    struct tag_bintree_node* right;
} bintree_node;
```

Ezt felhasználva csináljunk egy fejléc jellegű LZW fa struktúrát:

```
typedef struct
{
    bintree_node *root;
    int n_nodes;
```

```
    bintree_node *ins;
} lzw_hdr;
```

Írjunk pár függvényt hozzá:

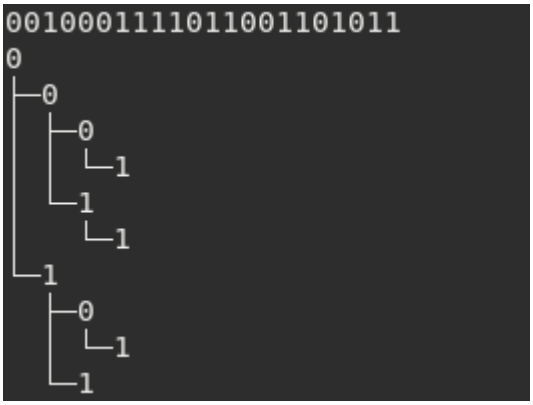
```
lzw_hdr lzw_create(void)
{
    lzw_hdr tree;
    tree.root = calloc(1, sizeof(bintree_node));
    tree.n_nodes = 1;
    tree.ins = tree.root;
    return tree;
}

void lzw_insert(lzw_hdr *tree, char ch)
{
    if(tree->root == NULL || tree->ins == NULL) return;

    bintree_node **child = NULL;

    if(ch == '0') child = &tree->ins->left;
    else if(ch =='1') child = &tree->ins->right;
    else return;

    if(*child == NULL)
    {
        *child = calloc(1, sizeof(bintree_node));
        //(*child)->data = tree->n_nodes++;
        (*child)->data = ch == '0' ? 0 : 1;

        tree->ins = tree->root;
    }
    else tree->ins = *child;
}

void lzw_free(lzw_hdr *tree)
{
    bintree_free(tree->root);
    tree->root = tree->ins = NULL;
    tree->n_nodes = 0;
}

void bintree_free(bintree_node *root)
{
    if(root->left != NULL)
        bintree_free(root->left);
    if(root->right != NULL)
        bintree_free(root->right);
    free(root);
```

```
}
```

És készen is van minden, amire szükségünk van LZW-fák létrehozásához. Egy egyszerű `main()`, hogy ki is tudjuk próbálni: (A `bintree_print()` implementációjához lásd a teljes forrást)

```
int main()
{
    srand(1);
    setlocale(LC_ALL, "");

    lzw_hdr tree = lzw_create();

    char input;
    while(!feof(stdin))
    {
        input = (char)getchar();
        if(input != '0' && input != '1') continue;

        lzw_insert(&tree, input);
    }

    bintree_print(tree.root, 0);

    lzw_free(&tree);

    return 0;
}
```

Futtassuk le a programot, adjunk neki bemenetet:



6.3. Fabejárás

Járd be az előző (inorder bejárású) fát pre- és posztorder is!

Megoldás forrása:

Rekuzió használatával néha kifejezetten nehéz problémákra is tudunk egyszerű megoldásokat találni. Az előző feladatban létrehozott bináris fa bejárása is lényegében négy rekurzív lépésben bejárható:

- megnézzük, hogy a gyökér csomópont `NULL` érték-e, ha igen, visszatérünk
- feldolgozzuk a csomópontot (ezesetben kiírjuk)
- bejárjuk a bal oldali részfát
- bejárjuk a jobb oldali részfát

Az utolsó három lépés sorrendje alapján kapunk preorder, inorder, vagy postorder bejárást. Ezek így néznek ki C-ben:

```
void bintree_preorder_print(bintree_node *root)
{
    if(root == NULL) return;
    printf("%d", root->data);
    bintree_preorder_print(root->left);
    bintree_preorder_print(root->right);
}

void bintree_inorder_print(bintree_node *root)
{
    if(root == NULL) return;
    bintree_preorder_print(root->left);
    printf("%d", root->data);
    bintree_preorder_print(root->right);
}

void bintree_postorder_print(bintree_node *root)
{
    if(root == NULL) return;
    bintree_postorder_print(root->left);
    bintree_postorder_print(root->right);
    printf("%d", root->data);
}
```

Ha a bejárás során tovább akarunk adni valamilyen adatot, például az eddigi lépések számát (a tényleges gyökértől való távolság), ezt megtehetjük egy plusz paraméter segítségével. Mivel C-ben nem adhatunk egy függvény paramétereinek alapértelmezett értéket, így a könnyebb használat érdekében két függvényt érdemes írnunk. Itt a preorder bejárás "továbbfejlesztett" változata:

```
void bintree_preorder_print_d(bintree_node *root)
{
    _bintree_preorder_print_d(root, 0);
}

void _bintree_preorder_print_d(bintree_node *root, int depth)
{
    if(root == NULL) return;
```

```
    for(int i = 0; i < depth; i++) printf(" ");
    printf("%d\n", root->data);
    _bintree_preorder_print_d(root->left, depth + 1);
    _bintree_preorder_print_d(root->right, depth + 1);
}
```

Az teljes forráskódban fellelhető `bintree_print()` függvény is tulajdonképpen ennek felel meg, annyi különbséggel, hogy szóközök helyett egy kevés extra logika alapján és egy pár Unicode karakter felhasználásával egy fa szerkezetet rajzol ki.

6.4. Tag a gyökér

Az LZW algoritmust ültesd át egy C++ osztályba, legyen egy Tree és egy beágyazott Node osztálya. A gyökér csomópont legyen kompozícióban a fával!

Megoldás videó: https://youtu.be/_mu54BDkqiQ

Megoldás forrása: ugyanott.

6.5. Mutató a gyökér

Írd át az előző forrást, hogy a gyökér csomópont ne kompozícióban, csak aggregációban legyen a fával!

Megoldás videó: https://youtu.be/_mu54BDkqiQ

Megoldás forrása: ugyanott.

Ez a feladat a kompozícióhoz képest annyiban tér el, hogy a

```
node root;
```

helyett

```
node *root;
```

fog szerepelni a `bin_tree<>` osztályban, illetve a konstruktorokban és a destruktorban `new` és `delete` utasításokra van szükségünk:

```
bin_tree() : root(new node()) {}
bin_tree(const T& r) : root(new node(r)) {}
~bin_tree() { delete root; }
```

Használhatunk ehelyett `std::unique_ptr<node>`-t is a birtoklás erősebb kifejezésére, ekkor külön `delete`-re sincs szükségünk.

6.6. Mozgató szemantika

Írj az előző programhoz másoló/mozgató konstruktort és értékadást, a mozgató konstruktor legyen a mozgató értékadásra alapozva, a másoló értékadás pedig a másoló konstruktorra!

Megoldás videó: https://youtu.be/QBD3zh5OJ0Y

Megoldás forrása: ugyanott.

Késztetést érzek rá, hogy a feladattól eltérjek, mivel a mozgató konstruktort a mozgató értékadásra alapozni elég szokatlan és előnytelen dolog (másolásnál is). Általában pont fordítva szokott lenni: a konstruktornak egyszerűbb a feladata, csak fel kell építenie egy példányt adott másolandó/mozgatandó példány alapján. Az értékadó operátoroknak viszont emellett meg is kell semmisíteniük az előző érték mögött álló példányt.

6.7. Vörös Pipacs Pokol/5x5x5 ObservationFromGrid

Megoldás forrása: https://github.com/nbatfai/RedFlowerHell

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

Írj Qt C++-ban egy hangyaszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Megoldás videó: https://bhaxor.blog.hu/2018/10/10/myrmecologist

Megoldás forrása: https://gitlab.com/nbatfai/bhax/-/tree/master/attention_raising%2FMyrmecologist

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

7.2. Java életjáték

Írd meg Java-ban a John Horton Conway-féle életjátékot, valósítsa meg a sikló-kilövőt!

Megoldás forrása: https://regi.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html#conway

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás forrása: https://bhaxor.blog.hu/2018/09/09/ismerkedes_az_eletjatekkal

Megoldás videó: a hivatkozott blogba ágyazva.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

7.4. BrainB Benchmark

Megoldás videó: initial hack: https://www.twitch.tv/videos/139186614

Megoldás forrása: https://github.com/nbatfai/esport-talent-search

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

7.5. Vörös Pipacs Pokol/19 RF

Megoldás videó: https://youtu.be/VP0kfvRYD1Y

Megoldás forrása: https://github.com/nbatfai/RedFlowerHell

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python (lehet az SMNIST [**?**] is a példa).

Megoldás videó: https://youtu.be/j7f9SkJR3oc

Megoldás forrása: https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0 (/tensorflow-0.9.0/tensorflow/exa
https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

8.2. Mély MNIST

Python (MNIST vagy SMNIST, bármelyik, amely nem a softmaxos, például lehet egy CNN-es).

Megoldás videó: https://bhaxor.blog.hu/2019/03/10/the_semantic_mnist

Megoldás forrása: SMNIST, https://gitlab.com/nbatfai/smnist

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

8.3. Minecraft-MALMÖ

Most, hogy már van némi ágensprogramozási gyakorlatod, adj egy rövid általános áttekintést a MALMÖ projektről!

Megoldás videó: initial hack: https://youtu.be/bAPSu3Rndi8. Red Flower Hell: https://github.com/nbatfai/-RedFlowerHell.

Megoldás forrása: a Red Flower Hell repójában.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

8.4. Vörös Pipacs Pokol/javíts a 19 RF-en

Megoldás forrása: https://github.com/nbatfai/RedFlowerHell

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó: https://youtu.be/z6NJE2a1zIA

Megoldás forrása: ugyanott.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

9.2. Gimp Scheme Script-fu: króm effekt

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely megvalósítja a króm effektet egy bemenő szövegre!

Megoldás videó: https://youtu.be/OKdAkI_c7Sc

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Chrome

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

9.3. Gimp Scheme Script-fu: név mandala

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely név-mandalát készít a bemenő szövegből!

Megoldás videó: https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelese_a_scheme_programozasi_nyelv

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Mandala

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

9.4. Vörös Pipacs Pokol/javíts tovább a javított 19 RF-eden

Megoldás forrása: https://github.com/nbatfai/RedFlowerHell

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

-számítógépes programozási nyelvek három szintje: gépi nyelv, assembly szintű nyelv, magas szintű nyelv -a könyv a magas szintű programozási nyelvek jellemzőiről, nyelvi eszközeiről, használatáról szól -a magas szintű nyelveken megírt programot forráskódnak nevezzük -forráskódot hardver közvetlenül nem tud futtatni, ehhez a forráskódot gépi nyelvbeli parancsokká kell alakítani -két módszer: fordítás, interpretálás (ezeket együttesen is lehet alkalmazni) -a magas szintű programozási nyelvek nem függnek az adott eszköz architektúrájától, így nem kell egy programot sokszor megírni, ezeknek a különbségeknek a kezelése a programok fordítására/értelmezésére szolgáló eszközkészletek feladata -minden programnyelvnél fontos a nyelv szintaktikai és szemantikai szabályainak pontos meghatározása -ezt hivatkozási nyelvnek hívjuk -az így meghatározott nyelv értelmezését adott platformon megvalósító programok a nyelv implementációi -a típus, mint absztrakt nyelvi eszköz: tartomány, műveletek, reprezentáció -új típusok definiálása -egyszerű és összetett adattípusok -mutató típusok -nevesített konstansok, változók

10.2. C programozás bevezetés

Brian Kernighan és Dennis Ritchie A C programozási nyelv című könyvének mindmáig igen meghatározó szerepe van, hiszen ezzel a könyvvel robbant be a köztudatba a C, mely ma is az egyik legelterjedtebb programnyelv. (Bár már sok nyelv megelőzte, melyek fordító/futtató környezetei viszont -ironikusan- sok esetben C-re alapulnak (pl. Python))

A könyv első kiadásában leírt nyelvi szabályok a C legkorábbi változatára vonatkoznak, erre a "szabványra" általában K&R C-ként hivatkoznak (ez az "eredeti C"). Ezt a szabványt a későbbiekben követték az ANSI C(/C89), a C99, a C11, és legújabban a C18 szabványok.

A könyv első fejezete egy egyszerű "Hello world!" program bemutatásával indul. Megemlítendő, hogy Kerninghan és Ritchie álltak elő először a "Helló Világ!" ötlettel, mely azóta is igen gyakran jelenik meg programnyelvek/környezetek első bemutatásaként. Tovább olvasva egyszerű példákon keresztül megismerhetjük a C nyelv által nyújtott főbb lehetőségeket: a változókat (és ezzel együtt a C típuskezelését), aritmetikai kifejezéseket, sztringeket, tömböket, szimbolikus állandókat, elágazásokat, ciklusokat, függvényeket, és az stdio.h könyvtárat. Mindezt a további fejezetekben részletezi a könyv.

Egy C program alapvető szerkezeti egységei a függvények. Mindaz a kód, ami fordítás során tényleges utasításokat eredményez, függvényekbe van foglalva, ezeken az egységeken kívül csak include-ok, illetve globális változók, függvények és szimbolikus állandók deklarációi szerepelnek. Egy függvénynek van visszatérési értéke, neve, argumentumlistája és teste, megyet egy kapcsoszárójelek közötti ún. blokkban helyezünk el (ezek a blokkok a vezérlő utasításoknál is fontosak lesznek). Minden függvényargumentum érték szerint kerül átadásra, cím szerinti ádatásra mutató típusok segítségével van lehetőségünk. C-ben, néhány más programnyelvvel ellentétben, a függvények és az eljárások nincsenek megkülönböztetve; ha azt akarjuk, hogy egy függvény ne térjen vissza semmilyen értékkel, a void speciális típust használjuk. Egy C program futása a main() függvénynél indul, ennek a függvénynek a megléte nélkül nem tudunk futtatható állományt létrehozni.

Egy program változói szigorúan rendelkeznek típussal. A változók típusa fontos szemantikai információt hordoz, hiszen a fordítóprogam ez alapján tudja eldönteni, hogy mekkora memóriaterületet tartson fenn egy értéknek, illetve, hogy miként kezelje a tárolt értékeket pl. aritmetikai utasításokban. Implicit (jelöletlen) módon csak néhány esetben végezhetünk értékadást illetve aritmetikai műveleteket eltérő típusú változók között, alapesetben ilyenkor jelölnünk kell a konverziót (cast). Megemlítendők a mutató típusok, melyek szemantikailag egy memóriacímet képviselnek, a tömbök, melyek több azonos típusú és logikailag összetartozó érték egyszerű tárolását teszik lehetővé, illetve a struktúrák, melyek adott esetben különböző típusú változók jól meghatározott sorozatát foglalják egy egységbe.

A C nyelv számos vezérlési szerkezetet nyújt a program futásának irányítására. Megjelennek a mára teljesen természetessé vált if - else if - else elágazások, a switch - case utasítások, a while és for cikusszervező utasítások, de van lehetőségünk, az Assembly világára emlékeztető módon, címkéket és ugrásokat (goto) használni (bár ezek gyakorlatban igen ritkán jelennek meg).

10.3. C++ programozás

Rövid olvasónapló a [BMECPP] könyvről.

10.4. Python nyelvi bevezetés

Rövid olvasónapló a [**?**] könyvről.

III. rész

Második felvonás

Bátf41 Haxor Stream
A feladatokkal kapcsolatos élő adásokat sugároz a https://www.twitch.tv/nbatfai csatorna, melynek permanens archívuma a https://www.youtube.com/c/nbatfai csatornán található.

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

Az előző feladat kódját fejleszd tovább: vizsgáld, hogy Vannak-e Java osztályok a Pi hexadecimális kifejtésében!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

11.4. C

[KERNIGHANRITCHIE] Kernighan, Brian W. Ã©s Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

11.5. C++

[BMECPP] Benedek, Zoltán Ã©s Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

11.6. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/-pdf/0404/0404335v7.pdf , 2004.

Köszönet illeti a NEMESPOR, https://groups.google.com/forum/#!forum/nemespor, az UDPROG tanulószoba, https://www.facebook.com/groups/udprog, a DEAC-Hackers előszoba, https://www.facebook.com/-groups/DEACHackers (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésükért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPROG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségben született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.